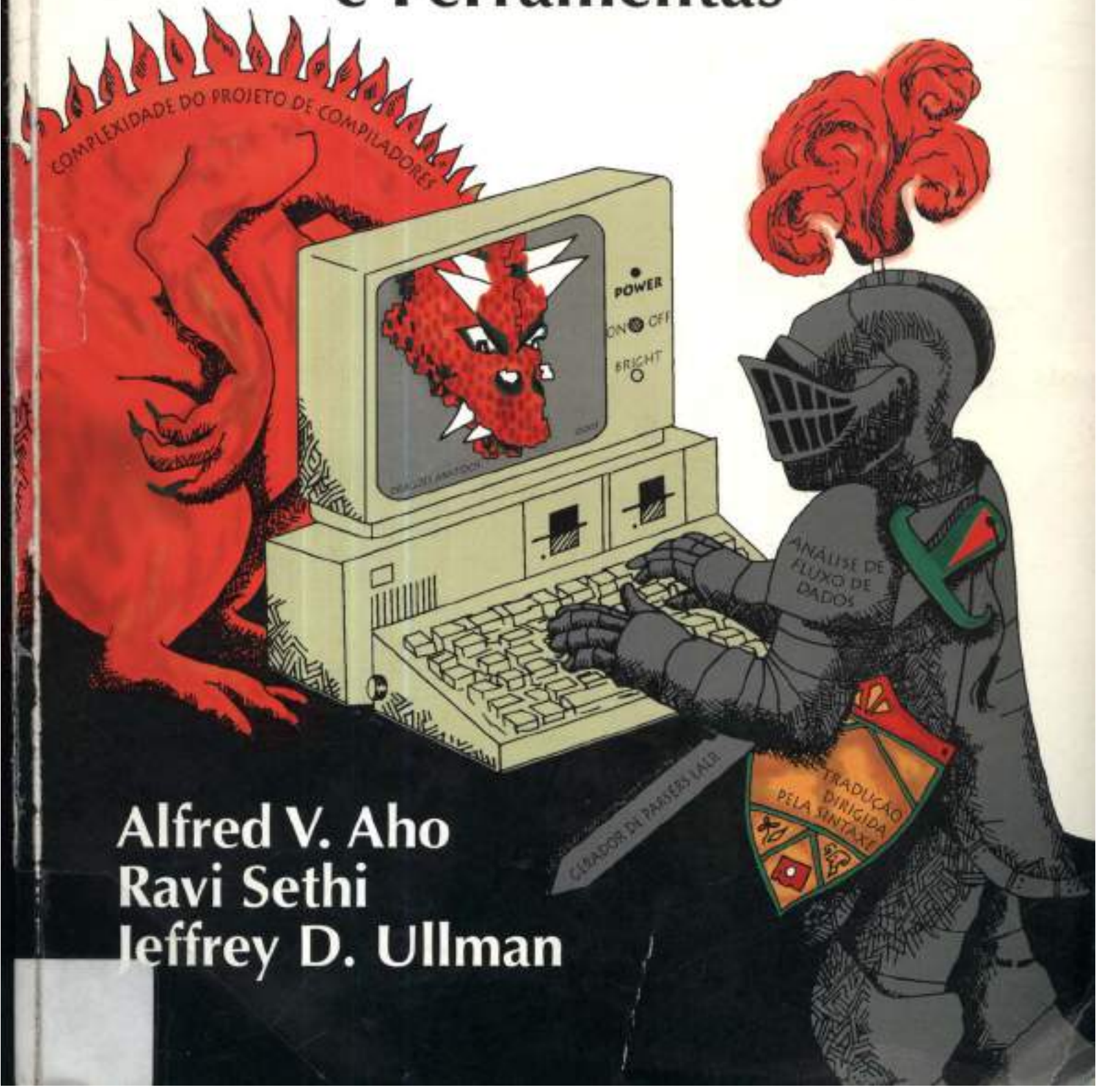
Compiladores

Princípios, Técnicas e Ferramentas



Alfred V. Aho
Ravi Sethi
Jeffrey D. Ullman

COMPILADORES

Princípios, Técnicas e Ferramentas

ALFRED V. AHO
AT & T Bell Laboratories
Murray Hill, New Jersey

RAVI SETHI
AT & T Bell Laboratories
Murray Hill, New Jersey

JEFFREY D. ULLMAN
Stanford University
Stanford, California

Tradução

DANIEL DE ARIOSTO PINTO

Professor Auxiliar no Departamento de Informática e Ciência da Computação
do Instituto de Matemática e Estatística da UERJ nas disciplinas: Compilação,
Linguagem Assembler e Estrutura de Linguagens de Programação
Tecnólogo em Processamento de Dados pela
Faculdade de Administração de Guanabara - FAG
Coordenador do Suporte de Sistemas do Banco de Dados
e Comunicação de Dados do IBGE
Bacharel em Administração de Empresas pela FAG

LTC
EDITORA

005.269

Computer Principles, Techniques and Tools
Copyright © 1986 by Bell Telephone Laboratories, Incorporated
Publicado originalmente nos Estados Unidos por Addison-Wesley
Publishing Company, Reading, Massachusetts, 01867 USA

Direitos exclusivos para a língua portuguesa
Copyright © 1995 by
LTC — Livros Técnicos e Científicos Editora S.A.
Travessa do Ouvidor, 11
Rio de Janeiro, RJ — CEP 20040-040
Tel.: (21) 221-9621
Fax: (21) 221-3202

Reservados todos os direitos. É proibida a duplicação ou
reprodução deste volume, no todo ou em parte,
sob quaisquer formas ou por quaisquer meios
(eletrônico, mecânico, gravação, fotocópia ou outros),
sem permissão expressa da Editora.

Prefácio

Este livro descende de outro, *Principles of Compiler Design*, de autoria de Alfred V. Aho e Jeffrey D. Ullman. Como seu antecessor, está destinado a ser um livro-texto para um primeiro curso em projeto de compiladores. A ênfase está na solução de problemas universalmente encontrados no projeto de um tradutor de linguagem, independentemente da máquina-forma ou alvo.

Embora poucas pessoas tenham a oportunidade de construir um mesmo manter um compilador de uma grande linguagem de programação, o leitor pode aplicar com proveito as idéias e técnicas discutidas neste livro ao projeto geral de programas. Por exemplo, as técnicas de reconhecimento de cadeias de caracteres para a construção de analisadores léxicos têm sido usadas em editores de texto, sistemas de recuperação de informações e programas de reconhecimento de padrões. As linguagens livres de contexto e as definições dirigidas pela sintaxe têm sido utilizadas na construção de muitas pequenas linguagens, tais como a de composição tipográfica e desenho de figuras, que produziram o original em inglês deste livro. Até as técnicas de otimização têm sido aproveitadas em verificadores de programas e em programas que produzem programas "estruturados" a partir dos não-estruturados.

Uso do Livro

Os tópicos mais importantes no projeto de compiladores são abordados em profundidade. O primeiro capítulo introduz a estrutura básica de um compilador e é essencial para o restante do livro.

O Capítulo 2 apresenta um tradutor de expressões infixas para a forma posfixa, construído a partir do uso de algumas das técnicas básicas descritas neste livro. Muitos dos capítulos restantes abordam em maiores detalhes os assuntos tratados no Capítulo 2.

O Capítulo 3 aborda a análise léxica, expressões regulares, máquinas finitas de estado e ferramentas de geração de *scanners*. As informações contidas nesse capítulo são amplamente aplicáveis ao processamento de textos.

O Capítulo 4 trata, de forma abrangente, das técnicas mais importantes de análise sintática, não-dos métodos de descendência recursiva, que são adequados à implementação manual, as técnicas LR, mais completas do ponto de vista computacional, e que têm sido usadas nos geradores de *parsers*.

O Capítulo 5 introduz as idéias primárias da tradução dirigida pela sintaxe. Esse capítulo é usado ao longo do livro, tanto para especificar quanto implementar traduções.

O Capítulo 6 apresenta as idéias principais para a realização da verificação estática e semântica de tipos. A verificação e a unificação de tipos são discutidas em detalhes.

O Capítulo 7 discute as organizações de memória usadas para dar suporte ao ambiente em tempo de execução de um programa.

O Capítulo 8 começa com uma discussão das linguagens intermediárias e mostra, então, como as construções comuns das linguagens de programação podem ser traduzidas em código intermediário.

O Capítulo 9 trata da geração do código-alvo. Estão incluídos os métodos básicos de geração de código "em vôo", bem como métodos simples para a geração de código para expressões. A otimização *peephole* e os geradores de geradores de código também são abordados neste capítulo.

O Capítulo 10 traz uma abordagem abrangente da otimização de código. Os métodos de análise de fluxo de dados aparecem bem detalhados, assim como os principais métodos de otimização global.

O Capítulo 11 discute alguns temas práticos que emergem na implementação de um compilador. As organizações deste capítulo são particularmente importantes na construção de compiladores.

O Capítulo 12 contém estudos de casos de compiladores construídos a partir de algumas das técnicas apresentadas neste livro.

O Apêndice A descreve uma linguagem simples, um "subconjunto" de Pascal que pode ser usada como base de um projeto de implementação.

Utilizando o conteúdo deste livro, os autores ministraram cursos introdutórios e avançados, em níveis de graduação e pós-graduação, nos AT & T Bell Laboratories, e nas Universidades de Columbia, Princeton e Stanford.

Um curso introdutório em compiladores poderia compreender as seguintes seções deste livro.

introdução	Capítulo 1 e Seções 2.1-2.5
análise léxica	2.6, 3.1-3.4
tabelas de símbolos	2.7, 7.6
análise sintática	2.4, 4.1-4.4
tradução dirigida pela sintaxe	2.5, 5.1-5.5
verificação de tipos	6.1-6.2
organização em tempo de execução	7.1-7.3
geração de código intermediário	8.1-8.3
geração de código	9.1-9.4
otimização de código	10.1-10.2

As informações necessárias a um projeto de programação, como o do Apêndice A, são introduzidas no Capítulo 2.

Um curso enfatizando as ferramentas na construção de compiladores deveria incluir a discussão dos geradores de analisadores léxicos da Seção 3.5, dos geradores de analisadores sintáticos das Seções 4.8 e 4.9, dos geradores de geradores de código da Seção 6.12, e o material sobre as técnicas de construção de compiladores encontrado no Capítulo 11.

Um curso variado deveria salientar os algoritmos usados nos geradores de analisadores léxicos e sintáticos discutidos nos Capítulos 3 e 4; a informação sobre a equivalência de tipos, sobrecarga, polimorfismo e unificação do Capítulo 6; o texto sobre organização de memória em tempo de execução do Capítulo 7; os métodos de geração de código dirigidos por padrões discutidos no Capítulo 9; e o material sobre otimização de código contido no Capítulo 10.

Exercícios

Como antes, classificamos os exercícios com estrelas. Os exercícios sem estrelas testam a compreensão das definições; aqueles com uma estrela dão origem a assuntos mais avançados, e os exercícios com duas estrelas são pequenos projetos de pesquisa.

Agradecimentos

Nas diversas etapas de elaboração deste livro, várias pessoas fizeram comentários valiosos sobre o texto. Nesse sentido, temos um débito de gratidão com Bill Appelbe, Nelson Beebe, Jon Bentley, Lois Bogess, Rodney Farrow, Stu Feldman, Charles Fischer, Chris Fraser, Art Gittelman, Eric Grosse, Dave Hanson, Eric Hoffman, Robert Henry, Gerard Holzmann, Steve Johnson, Brian Kernighan, Ken Kubota, Daniel Lehmann, Dave MacQueen, Dianne Mehl, Alan Martin, Doug McIlroy, Charles McLaughlin, John Mitchell, Elliott Organick, Robert Page, Phil Pfeiffer, Rob Pike, Kari Louko Rahn, Dennis Ritchie, Sriram Sankar, Paul Stoecker, Bjarne Stroustrup, Tom Szymanski, Kim Tracy, Peter Weinberger, Jennifer Widom e Reinhard Wilhelm.

Este livro foi composto pelos autores usando o excelente software disponível no sistema UNIX. O comando [illegible] de composição de tipos

[illegible]

é a linguagem em que Brian Kernighan desenvolveu para composição de figuras. Agradecemos especialmente a ele pela ajuda nos desenhos de figuras.

Destacamos ainda a linguagem [illegible] de Mike Lesk, para dispor tabelas, a [illegible], de Brian Kernighan e Lorinda Cherry, para a composição de tipos matemáticos, e o programa de Joe Ossanna, para a formatação de textos para uma fotocomposição de tipos, que em nosso caso foi um Mergenthaler Linotron 202/N. O gerador [illegible] de macros [illegible] foi escrito por Mike Lesk. Além disso, gostaríamos de agradecer [illegible] de autoria de Stu Feldman. As referências cruzadas no texto foram [illegible] criado por Al Aho, Brian Kernighan e Peter Weinberger, e [illegible] criado por Lee McMahon.

Os autores gostariam de agradecer particularmente a Patricia Solomon pelo preparo do texto para a fotocomposição. Sua motivação e competência no trabalho de digitação foram grandemente apreciadas. J. D. Ullman foi patrocinado pela Fundação Guggenheim, pela Universidade Stanford, pela Academia Israelense de Artes e Ciências durante parte do tempo de elaboração do livro. Por fim, os autores desejariam agradecer aos AT & T Bell Laboratories por seu apoio durante a preparação do texto.

A. V. A.
R. S.
J. D. U.

Sumário

Capítulo 1 Introdução à Compilação, 1

1.1 Compiladores, 1
1.2 Análise do programa-fonte, 2
1.3 As fases de um compilador, 5
1.4 Os aspectos do compilador, 8
1.5 O agrupamento das fases, 9
1.6 Ferramentas para a construção de compiladores, 10
Notas bibliográficas, 11

Capítulo 2 Um Compilador Simples de Uma Passagem, 12

2.1 Visão geral, 12
2.2 Definição da sintaxe, 12
2.3 Tradução dirigida pela sintaxe, 15
2.4 A análise gramatical, 18
2.5 Um tradutor para expressões simples, 21
2.6 Análise léxica, 25
2.7 Incorporando uma tabela de símbolos, 27
2.8 Máquinas de pilha abstratas, 28
2.9 Juntando as técnicas, 30
Exercícios, 36
Exercícios de programação, 36
Notas bibliográficas, 37

Capítulo 3 A Análise Léxica, 38

3.1 O papel do analisador léxico, 38
3.2 Bufferização da entrada, 40
3.3 Especificação dos tokens, 42
3.4 O reconhecimento de tokens, 45
3.5 Uma linguagem para especificação de analisadores léxicos, 48
3.6 Autômatos finitos, 51
3.7 De uma expressão regular para um AFN, 54
3.8 O projeto de um gerador de analisadores léxicos, 58
3.9 Otimização de reconhecedores de padrões baseados em AFDs, 60
Exercícios, 66
Exercícios de programação, 70
Notas bibliográficas, 70

Capítulo 4 Análise Sintática, 72

4.1 O papel do analisador sintático, 73
4.2 Gramáticas livres de contexto, 74
4.3 Escrevendo uma gramática, 77
4.4 Análise sintática *top-down*, 81
4.5 Análise sintática *bottom-up*, 86
4.6 Análise sintática de precedência de operadores, 90
4.7 Analisadores sintáticos LR, 94
4.8 Usando gramáticas ambíguas, 107
4.9 Geradores de analisadores sintáticos, 111
Exercícios, 115
Notas bibliográficas, 118

Capítulo 5 Tradução Dirigida pela Sintaxe, 120

5.1 Definições dirigidas pela sintaxe, 120
5.2 Construção de árvores sintáticas, 123
5.3 Avaliação *bottom-up* de definições s-atribuídas, 126
5.4 Definições bem-feitas, 127
5.5 Tradução *top-down*, 129
5.6 Avaliação *bottom-up* dos atributos herdados, 132
5.7 Avaliadores recursivos, 136
5.8 Espaço para os valores de atributos em tempo de compilação, 137
5.9 Reserva de espaço em tempo de construção do compilador, 138
5.10 Análise de definições dirigidas pela sintaxe, 141
Exercícios, 144
Notas bibliográficas, 145

Capítulo 6 Verificação de Tipos, 147

6.1 Sistemas de tipos, 147
6.2 Especificação de um verificador simples de tipos, 149
6.3 Equivalência das expressões de tipo, 151
6.4 Conversões de tipo, 153
6.5 Sobrecarga de funções e operadores, 154
6.6 Funções polimórficas, 156
6.7 Um algoritmo para unificação, 160
Exercícios, 163
Notas bibliográficas, 165

Capítulo 7 Ambientes em Tempo de Execução, 167

7.1 Temas da linguagem-fonte, 167

7.2 Organização de memória. 170
7.3 Estratégias para a alocação de memória. 172
7.4 Acessos a nomes não locais. 177
7.5 Transmissão de parâmetros. 183
7.6 Tabelas de símbolos. 185
7.7 Facilidades de linguagem para a alocação dinâmica de memória. 190
7.8 Técnicas de alocação dinâmica de memória. 191
7.9 Alocação de memória em Fortran. 192
Exercícios. 196
Notas bibliográficas. 198

Capítulo 8 Geração de Código Intermediário, 200

8.1 Linguagens intermediárias. 200
8.2 Declarações. 204
8.3 Comandos de atribuição. 207
8.4 Expressões booleanas. 211
8.5 Comandos de desvio múltiplo. 215
8.6 Retrocorreção. 216
8.7 Chamadas de procedimentos. 219
Exercícios. 220
Notas bibliográficas. 221

Capítulo 9 Geração de Código, 222

9.1 Temas no projeto de um gerador de código. 222
9.2 A máquina-alvo. 224
9.3 Gerenciamento de memória em tempo de execução. 226
9.4 Blocos básicos e grafos de fluxo. 229
9.5 Informações de uso subsequente. 231
9.6 Um gerador de código simples. 232
9.7 Alocação e atribuição de registradores. 235
9.8 Representação de blocos básicos sob a forma de GDAs. 237
9.9 Otimização peephole. 240
9.10 A geração de código a partir de GDAs. 242
9.11 Algoritmo de programação dinâmica para a geração de código. 246
9.12 Geradores de geradores de código. 248
Exercícios. 252
Notas bibliográficas. 253

Capítulo 10 Otimização de Código, 254

10.1 Introdução. 254
10.2 As principais fontes de otimização. 257
10.3 Otimização em blocos básicos. 261
10.4 Laços em grafos de fluxo. 262
10.5 Introdução à análise global de fluxo de dados. 264
10.6 Solução iterativa para as equações de fluxo de dados. 271
10.7 Transformações de melhoria de código. 276
10.8 Lidando com ponteiros. 283
10.9 Análise de fluxo de dados de grafos de fluxo estruturado. 286
10.10 Algoritmos de fluxo de dados eficientes. 293
10.11 Uma ferramenta para a análise de fluxo de dados. 297
10.12 Estimativas de Cpus. 303
10.13 Depuração simbólica de código otimizado. 307
Exercícios. 311
Notas bibliográficas. 314

Capítulo 11 Deseja Escrever um Compilador?, 316

11.1 Planejando um compilador. 316
11.2 Abordagens para o desenvolvimento dos compiladores. 317
11.3 O ambiente de desenvolvimento de compiladores. 318
11.4 Teste e manutenção. 319

Capítulo 12 Um Rápido Exame de Alguns Compiladores, 321

12.1 LCN, um pré-processador para computação de tipos matemáticos. 321
12.2 Compiladores para Pascal. 321
12.3 Os compiladores C. 322
12.4 Os compiladores Fortran 77. 322
12.5 O compilador Bliss/11. 324
12.6 O compilador otimizante Modula-2. 325

Apêndice A Um Projeto de Programação, 326

A.1 Introdução. 326
A.2 Estrutura do programa. 326
A.3 Síntaxe de um subconjunto de Pascal. 326
A.4 Convenções léxicas. 327
A.5 Exercícios sugeridos. 327
A.6 Evolução do interpretador. 328
A.7 Extensões. 328

Bibliografia, 329
Índice Alfabético, 339

CAPÍTULO 1

INTRODUÇÃO À COMPILAÇÃO

Os princípios e técnicas da construção de compiladores são aplicáveis a uma variedade muito mais ampla de tarefas do que a tradução de programas de computador. Na verdade, o estudo de compiladores é, em grande parte, o estudo de como gerenciar e transformar informação. Os princípios básicos de análise de linguagens, que são parte integrante de um compilador, são aplicáveis a uma ampla variedade de problemas. Os algoritmos e estruturas de dados desenvolvidos para construir um compilador também são importantes em muitas outras áreas da ciência da computação. A habilidade de compreender e aplicar as técnicas de compilação é útil para qualquer cientista da computação.

Os princípios e técnicas da construção de compiladores são aplicados pela maioria dos programas que leem, manipulam, traduzem ou executam outros programas. Muitos desses programas, como os editores de texto, os formatadores de texto, os montadores, os ligadores, os depuradores e os sistemas de prototipação, são ferramentas de desenvolvimento de software que fazem parte de um ambiente de programação. Outros, como os interpretadores de consulta a bancos de dados e os sistemas de processamento de pacotes, são componentes de sistemas de aplicação. Embora muitos desses programas não precisem, tecnicamente falando, ser compilados, a maioria deles pode se beneficiar de muitas das técnicas desenvolvidas para a construção de compiladores. Além disso, muitos deles são complexos o suficiente para que sua implementação também se beneficie de muitas das técnicas de engenharia de software discutidas neste livro.

Nosso conhecimento sobre como organizar e escrever compiladores aumentou bastante desde que os primeiros compiladores foram escritos, no início dos anos 50. É difícil fornecer uma data exata para o primeiro compilador, porque muitos projetos iniciais tiveram uma grande qualidade exploratória e implementações foram realizadas independentemente por diversos grupos. Muitos dos trabalhos iniciais em compilação lidavam com a tradução de fórmulas aritméticas em código de máquina.

Na década dos anos 50, os compiladores foram considerados programas notoriamente difíceis de se escrever. O primeiro compilador de FORTRAN, por exemplo, consumiu 18 homens-ano para ser implementado [Backus et al., 1957]. Desde então, técnicas sistemáticas para o tratamento de muitas das importantes tarefas que ocorrem durante a compilação foram desenvolvidas, e implementações de linguagens de programação em ambientes de programação eficientes são agora relativamente fáceis de serem escritas. Mesmo assim, ainda um compilador ocasional é escrito para um projeto de estudantes, uma vez que a construção de compiladores continua sendo um excelente exercício de projeto de software.

1.1 COMPILADORES

Basicamente, um compilador é um programa que lê um programa escrito em uma linguagem — a linguagem *fonte* — e o traduz num programa equivalente numa outra linguagem — a linguagem *alvo* (veja a Fig. 1.1). Como parte importante deste processo de tradução, o compilador informa seu usuário da presença de erros no programa fonte.

À primeira vista, a complexidade de um compilador pode parecer assustadora. Existem milhares de linguagens fonte, que vão das linguagens de programação tradicionais, tais como FORTRAN e Pascal, às linguagens especializadas que emergiram virtualmente em todas as áreas de aplicação de computadores. As linguagens-alvo são igualmente variadas; uma linguagem-alvo pode ser uma outra linguagem de programação ou a linguagem de máquina de qualquer computador entre um microprocessador e um supercomputador. Os compiladores são algumas vezes classificados como de uma passagem, de passagens múltiplas, de otimização ou de depuração. Apesar dessa terminologia ser antiquada, ela reflete bem as preocupações que você encontrará ao estudar compiladores.

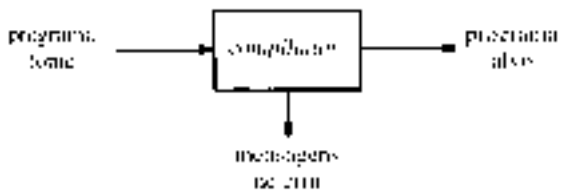


Fig. 1.1 Um compilador.

Você também deve estar ciente de que a construção de um compilador correto é uma tarefa difícil. Isso não significa que todos os compiladores precisam ser perfeitos, mas, como acontece com outros programas que lidam com informações complexas, os compiladores tendem a ser melhorados ao longo do tempo, à medida que problemas são descobertos. Felizmente, a engenharia de software avançou bastante nos últimos anos, tornando mais fácil a construção de compiladores confiáveis.

O Modelo de Compilação de Análise e Síntese

Existem duas partes no processo de compilação: a análise e a síntese. A parte de análise divide o programa fonte em suas partes constituintes e cria uma representação intermediária do programa fonte. A parte de síntese constrói o programa alvo desejado a partir da representação intermediária. Durante a síntese, operações específicas da máquina-alvo são determinadas. Normalmente, a análise é a parte mais bem compreendida da compilação; portanto, começaremos com ela. Também discutiremos a síntese de código, mas a ênfase será dada às técnicas que são independentes da máquina-alvo. Uma discussão mais detalhada sobre a síntese de código será feita no Capítulo 9.

Durante a análise, as operações implicadas pelo programa fonte são determinadas e registradas numa estrutura hierárquica, chamada de árvore. Frequentemente, é utilizada uma forma especial de árvore, chamada árvore sintática, na qual cada nó representa uma operação e os filhos de um nó representam os argumentos da operação. Por exemplo, a árvore sintática para uma atribuição de velocidade é mostrada na Fig. 1.2.

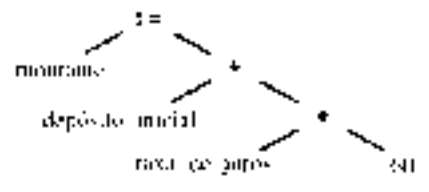


Fig. 1.2 Árvore sintática para [illegible]

Muitas características de uma linguagem de programação podem ser descritas por meio de uma gramática livre de contexto. Muitos exemplos de tais características são:

1. *Definição de variáveis.* Uma declaração como `int i, j, k;` em C pode ser descrita por uma gramática livre de contexto. Entretanto, uma gramática livre de contexto não é capaz de descrever a exigência de que uma variável deva ser declarada antes de ser usada. Mais ainda, a gramática livre de contexto não é capaz de descrever a exigência de que uma variável seja declarada no máximo uma vez. Essas características são descritas por meio de regras semânticas, que serão estudadas no Capítulo 6. Para descrever a declaração de variáveis, é preferível usar uma gramática livre de contexto e regras semânticas, em vez de uma gramática mais complexa que descreva todas as características de uma só vez.

2. *Verificação de tipos.* Uma expressão como `x + 1` pode ser válida ou não, dependendo do tipo da variável `x`. Por exemplo, se `x` for um vetor, a expressão não será válida. Se `x` for um registro, a expressão será válida se o registro contiver um campo inteiro chamado `x`. Se `x` for um inteiro, a expressão será válida. Regras semânticas são usadas para verificar se uma expressão é válida de acordo com o tipo de seus componentes. Essas regras semânticas também são descritas no Capítulo 6.

3. *Correspondência de parâmetros.* Em muitas linguagens de programação, o número de parâmetros reais numa chamada de procedimento deve ser igual ao número de parâmetros formais na declaração do procedimento. Além disso, em algumas linguagens, como Pascal, os tipos dos parâmetros reais devem ser iguais aos tipos dos parâmetros formais correspondentes. Essas características também são descritas por meio de regras semânticas.

4. *Geração de código.* Um compilador precisa gerar código para cada construção de uma linguagem de programação convencional. Portanto, as características mais complexas de uma linguagem de programação são representadas pelas construções que geram código. Por exemplo, a construção `while` em Pascal é compilada gerando-se código para testar a condição e desviar para fora do laço, se a condição for falsa. Veja Aho et al. [1986] e Ullman [1984] para uma discussão da compilação de `while`.

5. *Interpretação de operações.* Uma interpretação de operação é um trecho de código que implementa uma operação definida numa linguagem de programação. Por exemplo, uma operação de multiplicação de matrizes pode ser definida numa linguagem de programação, mas a implementação dessa operação pode ser deixada para uma rotina de biblioteca.

O Contexto de um Compilador

Além do próprio compilador, vários outros programas podem ser necessários para se criar um programa executável. Um programa fonte pode ser dividido em módulos armazenados em arquivos separados. A tarefa de coletar esses módulos fonte pode ser encarregada, pelo menos em parte, a um programa separado, chamado pré-processador. O programa fonte, após ser pré-processado, pode ser compilado. O programa objetivo gerado pelo compilador pode exigir que outros programas sejam executados antes que ele possa ser executado em uma máquina. O compilador pode gerar uma montagem em linguagem de montagem, que é então processada por um montador. Alternativamente, o compilador pode gerar código para ser interpretado por um interpretador.

Em sistemas de computação mais complexos, em vez de um programa fonte, um sistema pode ter de compilar uma coleção de definições de dados. Por exemplo, um sistema de gerência de banco de dados pode precisar compilar definições de esquemas, escritas numa linguagem de definição de dados, antes de compilar programas que acessem o banco de dados. Outro exemplo é um sistema de processamento de textos que precisa compilar uma descrição de formatação de documentos antes de formatar um documento específico.

Um compilador pode ser construído à mão ou gerado automaticamente a partir de uma especificação de linguagem. Em ambos os casos, a tecnologia de compilação envolve o uso de muitas ferramentas de software. Muitas dessas ferramentas são descritas neste livro. Alguns exemplos de tais ferramentas são dados a seguir:

1. *Analisadores sintáticos.* Um analisador sintático é um programa que determina a estrutura sintática de um programa fonte. Essa estrutura é usada em muitas fases subseqüentes da compilação. Muitas linguagens de programação podem ser descritas naturalmente por gramáticas livres de contexto, e existem algoritmos eficientes para se construir analisadores sintáticos a partir de gramáticas livres de contexto. Muitos desses algoritmos são descritos no Capítulo 4. Ferramentas para a geração de analisadores sintáticos são descritas no Capítulo 5.

2. *Analisadores semânticos.* Um analisador semântico é um programa que determina o significado de um programa fonte além de sua estrutura sintática. Por exemplo, um analisador semântico pode determinar se uma expressão é válida e se seus operandos são compatíveis. Algumas ferramentas para a geração de analisadores semânticos são descritas no Capítulo 6.

3. *Otimizadores de código.* Um otimizador de código é um programa que melhora o código gerado por um compilador. Algumas técnicas de otimização são descritas no Capítulo 10.

4. *Geradores de código.* Um gerador de código é um programa que gera código de máquina ou código de montagem a partir de uma representação intermediária do programa fonte. Um gerador de código pode ser construído a partir de uma descrição da máquina objetivo e de uma descrição de como traduzir cada construção da linguagem fonte em código para a máquina objetivo. Um gerador de código é descrito no Capítulo 9.

5. *Ferramentas de escrita.* Um compilador precisa produzir mensagens de erro e outras informações de escrita. Algumas ferramentas para a geração de interfaces de escrita são descritas no Capítulo 8.

6. *Ferramentas de depuração.* Um compilador pode ser depurado com a ajuda de ferramentas que permitem a execução passo a passo de um programa e a inspeção de seus dados. Algumas ferramentas para a geração de depuradores são descritas no Capítulo 8.

7. *Ferramentas de manutenção.* Um compilador pode ser mantido com a ajuda de ferramentas que mantêm a consistência de módulos fonte. Algumas ferramentas para a geração de ferramentas de manutenção são descritas no Capítulo 8.

1.2 ANÁLISE DO PROGRAMA FONTE

Nesta seção, introduzimos a análise e descrevemos algumas linguagens de programação fonte. O material aqui apresentado é detalhado nos Capítulos 3 e 4. Na compilação, a análise consiste em três fases:

1. *Análise linear*, na qual um fluxo de caracteres que constitui o programa fonte é lido da esquerda para a direita e agrupado em *tokens*, que são seqüências de caracteres com um significado coletivo.

2. *Análise hierárquica*, na qual os caracteres ou tokens são agrupados hierarquicamente em coleções com significado coletivo.

3. *Análise semântica*, na qual certas verificações são realizadas para assegurar que os componentes de um programa se ajustem de uma forma significativa.

1.2 Muitas das características de uma linguagem de programação podem ser descritas por meio de uma gramática livre de contexto. Muitos exemplos de tais características são dados a seguir:

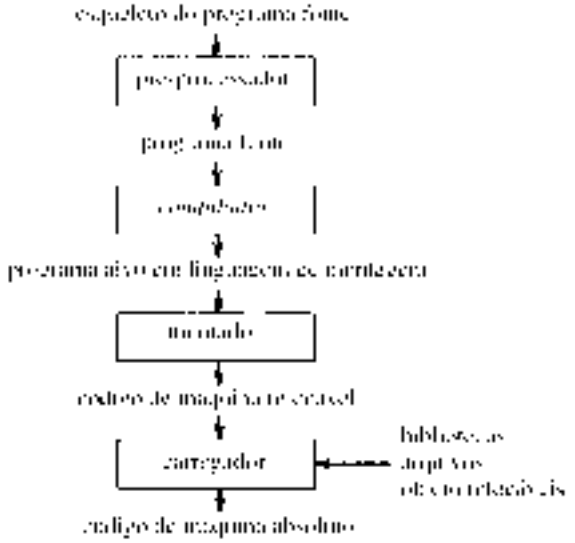


Fig. 1.3 Uma estrutura típica para sistemas de processamento de linguagens.

Análise Léxica

Para um compilador, a análise léxica é chamada de *análise linear* porque ela lê o programa fonte da esquerda para a direita, caractere por caractere. Por exemplo, na análise léxica de um comando de atribuição

```
posicao := posicao + velocidade * 60
```

poderíamos ser confrontados com as seguintes tarefas:

1. O identificador `posicao`.
2. O símbolo de atribuição `:=`.
3. O identificador `posicao`.
4. O sinal de adição `+`.
5. O identificador `velocidade`.
6. O sinal de multiplicação `*`.
7. O número `60`.

Os espaços que separam os caracteres desses tokens são normalmente eliminados durante a análise léxica.

Análise Sintática

A análise sintática é chamada de *análise gramatical* ou *análise sintática*. Envolve o agrupamento dos tokens do programa fonte em frases gramaticais que são usadas pelo compilador para sintetizar a saída. Usualmente, as frases gramaticais do programa fonte são representadas por uma árvore gramatical tal como a mostrada na Fig. 1.4.

Na expressão [illegible] a análise [illegible] sintática, a unidade lógica, porque as convenções usuais das expressões aritméticas nos dizem que a multiplicação é realizada antes da adição. A árvore de expressão [illegible] mostra a ordem natural dessas operações.

A estrutura hierárquica de um programa usualmente é expressa por regras recursivas. Por exemplo, podemos ter as seguintes regras como parte da definição de expressões:

1. Qualquer *identificador* é uma expressão.
2. Qualquer *número* é uma expressão.
3. Se $expressão_1$ e $expressão_2$ são expressões, então também o são

$$expressão_1 + expressão_2$$
$$expressão_1 * expressão_2$$
$$(expressão_1)$$

As regras 1 e 2 são as regras básicas (não recursivas), enquanto a regra 3 define expressões em termos dos operadores aplicados a outras expressões. Então, pela regra 1, [illegible] são expressões. Pela regra 2, [illegible] é uma expressão, enquanto que pela regra 3 podemos primeiro inferir que [illegible] é uma expressão e finalmente que [illegible] também o é.

Similarmente, muitas linguagens definem recursivamente coisas tais como:

1. Se *identificador* é um identificador e $expressão_1$ é uma expressão, então

$$identificador := expressão_1$$

é um comando.

2. Se $expressão_1$ é uma expressão e $comando_1$ um comando, então

while $expressão_1$ **do** $comando_1$
if $expressão_1$ **then** $comando_1$

são comandos.

A divisão entre conhecer sintaxe e semântica é muito arbitrária. Usualmente, esse livro coloca [illegible] simplesmente [illegible] da gramática

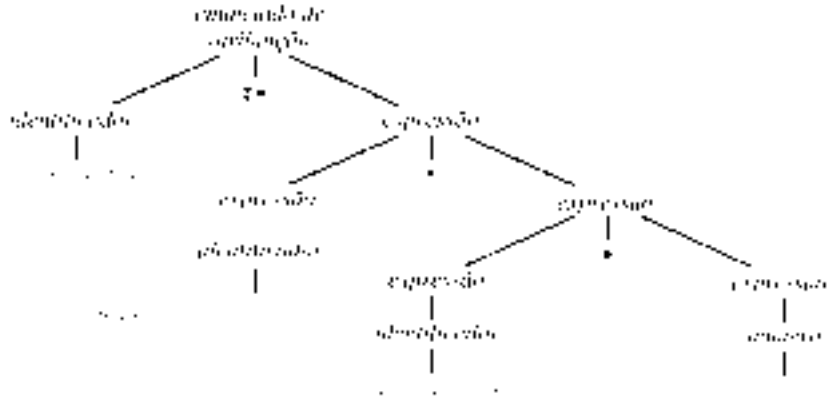


Fig. 1.4 Árvore gramatical para [illegible]

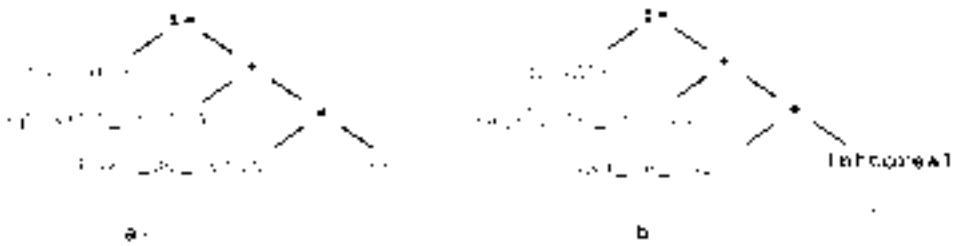


Fig. 1.5 Análise semântica: inserir uma conversão de inteiro para real

sintaxe. Um fator determinante na decisão é o de uma descrição da linguagem fonte ser inerentemente recursiva ou não. As construções iterativas não requerem recursão, enquanto que as estruturas frequentemente a exigem. As gramáticas livres de contexto são uma formalização das regras recursivas que podem ser usadas para guiar a análise sintática. São estudadas no Capítulo 2 e extensivamente estudadas no Capítulo 4.

Por exemplo, a recursão não é requerida para reconhecer identificadores que são tipicamente cadeias de letras e dígitos, começando por uma letra. Reconheceríamos normalmente os identificadores por um simples esquadrinhamento do fluxo de entrada, ignorando até que um caractere que não uma letra ou um dígito fosse encontrado, e agrupando esses caracteres lidos (junto com as informações sobre a posição daquele ponto). Os caracteres, dessa forma agrupados, seriam registrados numa tabela chamada tabela de símbolos, e removidos da entrada, de tal forma que o processamento do próximo *token* pudesse se iniciar.

Por outro lado, esse tipo de esquadrinhamento linear não é poderoso o suficiente para analisar expressões aritméticas. Por exemplo, não podemos fazer corresponder apropriadamente os parênteses nas expressões, estes [illegible] aninhados sem um algum tipo de estrutura hierárquica ou aninhamento na entrada.

A árvore gramatical da Fig. 1.4 descreve a estrutura sintática da entrada. Uma representação alternativa dessa estrutura sintática é dada pela árvore sintática na Fig. 1.5(a). Uma árvore sintática é uma representação condensada da árvore gramatical na qual os operadores figuram como nós internos e os operandos de um operador são os filhos desse operador. A construção de árvores, tais como aquela da Fig. 1.5(a), é discutida na Seção 5.2. Iremos examinar no Capítulo 2, e com mais detalhes no Capítulo 5, o tema da *tradução dirigida pela sintaxe*, na qual o compilador usa a estrutura hierárquica da entrada a fim de auxiliar a geração da saída.

Análise Semântica

A fase de análise semântica verifica os erros semânticos no programa fonte e coleta as informações de tipo para a fase subsequente de geração de código. Utiliza a estrutura hierárquica determinada pela fase de análise sintática, a fim de identificar os operadores e operandos das expressões e enunciados.

Um importante componente da análise semântica é a verificação de tipos. Nela o compilador checa se cada operador recebe os operandos que são permitidos pela especificação da linguagem fonte. Por exemplo, muitas definições nas linguagens de programação requerem que o compilador crie um aviso a cada vez que um número real seja usado para indexar um vetor. No entanto, a especificação da linguagem pode permitir algumas coerções de operandos, como por exemplo, quando um operador aritmético binário é aplicado a um inteiro e a um real. Nesse caso, o compilador pode precisar converter o inteiro para real. A verificação de tipos e a análise semântica serão estudadas no Capítulo 6.

Exemplo 1.1. Dentro da máquina, um padrão de bits representando um inteiro é geralmente diferente do padrão de bits para um real, mesmo quando o inteiro tem o mesmo valor que o número real (exibido como saída). Suponha, por exemplo, que todos os identificadores na Fig. 1.5 tenham sido declarados como reais e assuma que 60, por si só, seja um inteiro. A verificação de tipos da Fig. 1.5(a) revela que o + está aplicado a um real, [illegible] e a um inteiro, 60. Presuponha que o inteiro seja convertido em real. Isso foi conseguido na Fig. 1.5(b) pela criação de um nó extra para o operador **inttoreal**, que converte explicitamente um inteiro num real. Alternativamente, como o operando de **inttoreal** é uma constante, o compilador pode, em lugar, substituir a constante inteira por uma constante real equivalente.

A Análise nos Formatadores de Texto

É útil considerar a entrada para um formatador de texto como especificando uma hierarquia de *compartimentos*, que são regiões retangulares a serem preenchidas por algum padrão de *bits*, representando palavras a serem impressas num dispositivo de saída.

Por exemplo, o sistema TEX [Knuth 1984a] vê sua entrada dessa forma. Cada caractere, que não seja parte de um comando, representa um compartimento contendo o padrão de *bits* para aquele caractere na fonte e tamanho apropriados. Caracteres consecutivos não separados por espaços em branco ou espaços em caracteres de avanço de linha são agrupados em palavras, consistindo em uma sequência de compartimentos horizontalmente arranjados, mostrados esquematicamente na Fig. 1.6. O agrupamento de caracteres em palavras (ou comandos) é suspenso dentro de um [illegible] formatador de texto.

Os compartimentos em TEX podem ser construídos a partir de outros menores, através de combinações arbitrárias horizontais e verticais. Por exemplo,

[illegible]

agrupa a lista de compartimentos pela justaposição dos mesmos horizontalmente, enquanto que o operador [illegible] pode agrupar uma lista de compartimentos por justaposições vertical. Dessa forma, se escreveria em TEX

[illegible]

obtendo o arranjo de compartimentos mostrado na Fig. 1.7. A determinação do arranjo hierárquico de compartimentos estabelecido pela entrada é parte da análise sintática em TEX.

Como outro exemplo, o pré-processador EQN para a matemática [Kernighan e Cherry 1975] (um pré-processador oficialmente usado em TEX) constrói expressões matemáticas a partir de operadores como [illegible]

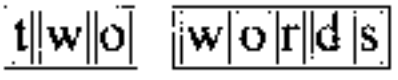


Fig. 1.6 Agrupamento de caracteres e palavras em *compartimentos*

Fig. 1.7 Um esquema de composição em TEX

e os indicadores subscritos e sobrescritos. No EQN encontra-se uma sequência de entrada da forma

BOX ... *box*

em que a cadeia de *box* é substituída por *BOX*, possuindo as características tipográficas indicadas na Fig. 1.8. O operador *sup* similarmente produz texto em caixa superior direita.

Esses operadores podem ser aninhados recursivamente, de tal forma que o texto EQN

...

resulta em ... O agrupamento dos operadores em expressões é parte da análise léxica do texto EQN. No entanto, é necessária a sua estrutura sintática para detectar erros sintáticos e a localização de um componente.

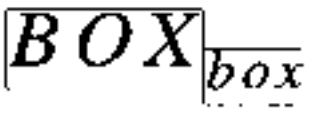


Fig. 1.8 Construção de uma estrutura de caixa para o texto matemático

1.3 AS FASES DE UM COMPILADOR

Conceitualmente, um compilador opera em *fases*, cada uma das quais transforma o programa fonte de uma representação para outra. Uma decomposição típica de um compilador é mostrada na Fig. 1.9. Na prática, algumas das fases podem ser agrupadas e a representação intermediária entre as duas não precisa ser explicitamente construída.

As três primeiras fases, formando o núcleo da parte de análise do compilador, foram anteriormente estudadas na última seção. Duas outras atividades, o gerenciamento da tabela de símbolos e a manipulação de erros, são mostradas interagindo com as seis fases de análise léxica, análise sintática, análise semântica, geração de código intermediário, otimização e geração de código. Informalmente, também chamaremos de "fases" o gerenciador da tabela de símbolos e o manipulador de erros.

Gerenciamento da Tabela de Símbolos

Uma função essencial de um compilador é registrar os identificadores usados no programa fonte e coletar as informações sobre os seus diversos atributos. Esses atributos podem prover informações sobre a memória reservada para o identificador, seu tipo, escopo (onde é válido no programa) e, no caso de nomes de procedimentos, coisas tais como o número e os tipos de seus argumentos, o método de transmissão de cada um (por exemplo, por referência) e o tipo retornado, se algum.

Uma *tabela de símbolos* é uma estrutura de dados contendo um registro para cada identificador, com os campos contendo os atributos do identificador. A estrutura de dados nos permite encontrar rapidamente o registro de cada identificador e, igualmente, armazenar ou recuperar dados rapidamente dele. As tabelas de símbolos são discutidas nos Capítulos 2 e 7.

Quando, no programa fonte, o analisador léxico detecta um identificador, ele o instala na tabela de símbolos. No entanto, os atributos do identificador não podem ser normalmente determinados durante a análise léxica. Por exemplo, em Pascal, numa declaração como

...

o tipo real ainda não será conhecido quando ... forem encontrados pelo analisador léxico.

As fases restantes coletam informações sobre os identificadores na tabela de símbolos e com elegância assumem diversas maneiras. Por exemplo, ao realizar a análise semântica e a geração de código intermediário, precisamos conhecer de que tipos os identificadores são, a fim de verificar se o programa fonte os usa de forma válida e, por conseguinte, gerar as operações apropriadas sobre os mesmos. O gerador de código tipicamente insere e usa as informações detalhadas a respeito dos nomes armazenadas nos identificadores.

Detecção de Erros e Geração de Relatórios

Cada fase pode encontrar erros. Entretanto, após encontrá-los, precisa tratar de alguma forma com os mesmos, de tal forma que a compilação possa continuar, permitindo que erros adicionais sejam detectados e, possivelmente, todos os erros posteriores do programa fonte. Um compilador que para ao encontrar o primeiro erro é tão prestativo quanto poderia ser.

As fases de análise sintática e semântica tratam usualmente de uma grande parte dos erros detectáveis pelo compilador. A fase de análise léxica pode detectá-los quando os caracteres remanescentes na entrada não formem nenhum token válido da linguagem. Os erros sintáticos são, de forma eficiente, detectados pelo analisador sintático. Durante a análise semântica o

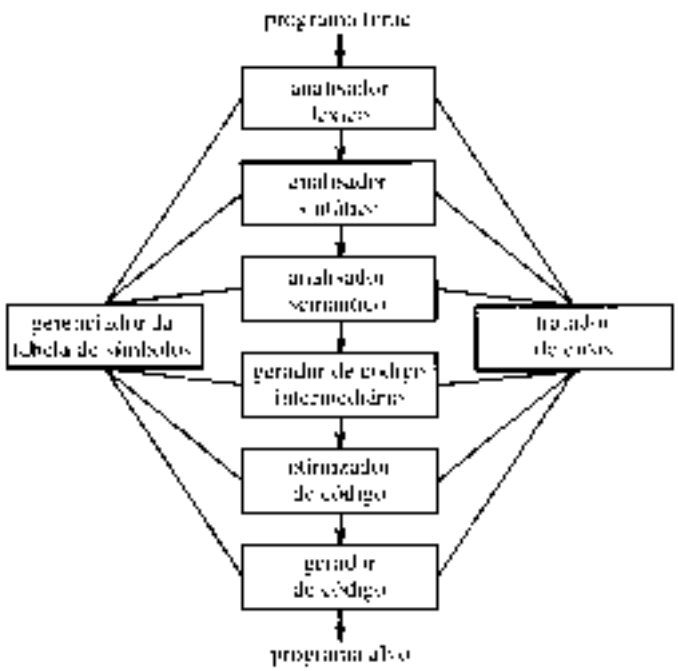


Fig. 1.9 Fases de um compilador

[illegible] sobre as construções que possuem [illegible] sobre as construções que possuem [illegible] significado da operação [illegible] por exemplo, [illegible] identificados, [illegible] identificador, uma palavra-chave [illegible] de ponto e vírgula [illegible] A seguir [illegible] para aquele [illegible]

Outros tokens [illegible] por que "[illegible]". Por exemplo, [illegible] mais [illegible] apenas [illegible] tabela de símbolos, [illegible] Os [illegible] aponta para a entrada [illegible] na tabela de símbolos.

[illegible]

As Fases de Análise

[illegible]

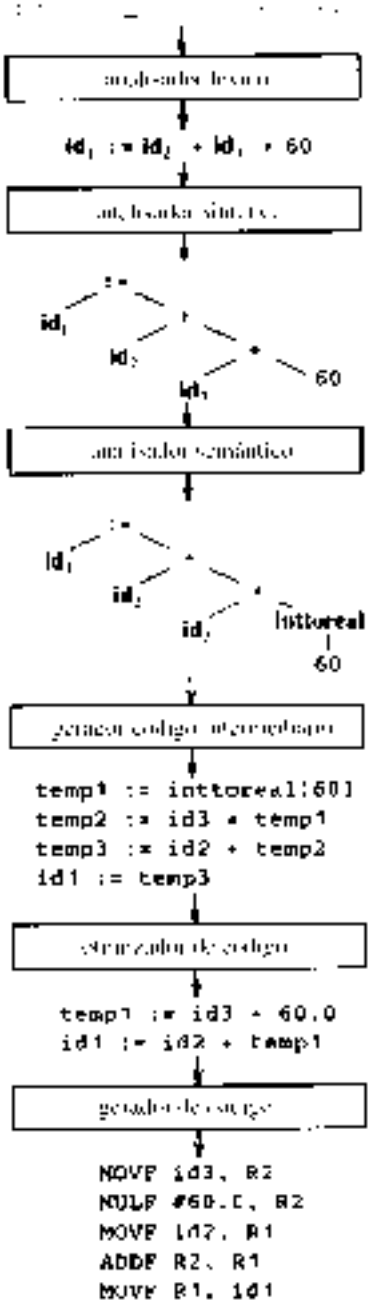


Fig. 1.10 Tradução de [illegible]

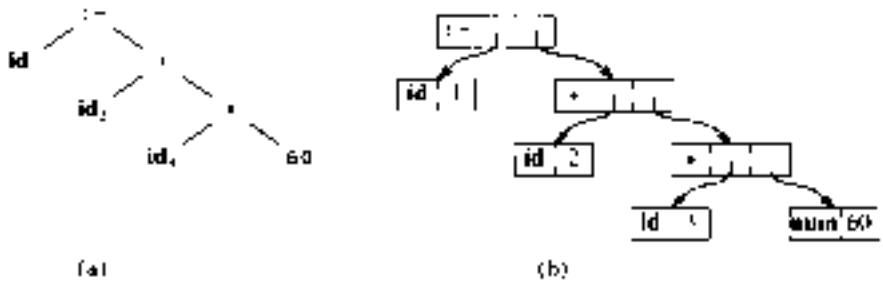


Fig. 1.11. Árvore de análise sintática e árvore sintática para a atribuição posicional := inicial + velocidade * 60.

representação intermediária. Depois, ele verifica e executa as instruções que formam o código intermediário. A representação de IT depois é traduzida em uma sequência de instruções de máquina, possivelmente otimizada por uma segunda passagem. Por exemplo, a árvore sintática da Figura 1.11(b) pode ser traduzida pela sequência de instruções

```
id1 := id2 + id3 * 60                                   (1.2)
```

Discutiremos mais detalhes sobre a representação intermediária, o código intermediário e as técnicas para sua tradução e execução mais adiante neste capítulo, começando na Seção 1.2. Muitos compiladores usam uma representação intermediária explícita, como a árvore sintática e a sequência de instruções da Figura 1.11, mas podem armazenar apenas as informações necessárias em uma tabela de símbolos para acessar mais tarde durante a geração de código.

A fase de análise é descrita em detalhes nos Capítulos 2 e 4. Nela, a análise sintática e a análise semântica são implementadas juntas. A análise sintática é baseada em uma gramática livre de contexto, que é uma notação recursiva poderosa o suficiente para descrever a estrutura aninhada da maioria das construções de linguagens de programação. Por exemplo, a gramática

```
expr → expr + expr | expr * expr | ( expr ) | id | num
```

descreve expressões aritméticas simples, onde `expr` representa uma expressão, `id` representa um identificador e `num` representa uma constante numérica. Uma regra semântica associada a uma produção como `expr → expr + expr` especifica que a soma dos dois subtermos `expr` representa uma expressão. Os Capítulos 2 e 4 apresentam algoritmos para a análise sintática e a análise semântica, respectivamente. Técnicas de tradução dirigida por sintaxe também são discutidas na Seção 2.3. Vamos ver também como a teoria de autômatos finitos é útil para descrever os componentes léxicos mais simples de uma linguagem, como a estrutura de identificadores e números. Essas técnicas são apresentadas na Seção 3.3. Em geral, os componentes léxicos são descritos com uma notação baseada em expressões regulares, e existem ferramentas para gerar automaticamente um reconhecedor para cada linguagem a partir de uma descrição dessa natureza. Autômatos finitos e expressões regulares aparecem na Figura 1.3. Eles são discutidos em detalhes na Seção 3.3. Ferramentas como o Lex e o Yacc são discutidas na Seção 3.5.

Geração de Código Intermediário

Após a análise sintática e a análise semântica, alguns compiladores geram uma representação intermediária explícita do programa fonte. Podemos pensar nessa representação intermediária como um programa para uma máquina abstrata. Essa representação intermediária deve ter duas propriedades importantes: ser fácil de produzir e fácil de traduzir para o programa-objeto.

A representação intermediária pode ter uma variedade de formas. No Capítulo 8, consideramos uma forma intermediária chamada código de três endereços, que é como uma montagem para uma máquina na qual cada posição de memória pode atuar como um registrador. O código de três endereços consiste em uma sequência de instruções, cada uma das quais tem no máximo três operandos. O programa na Figura 1.11(b) pode ser expresso no código de três endereços como

```
t1 := inttofloat(60)
t2 := id3 * t1
t3 := id2 + t2
id1 := t3                                               (1.3)
```

Há várias outras formas que a representação intermediária pode assumir. Existem representações em árvore, como as árvores sintáticas, e também representações lineares, como a notação polonesa inversa, que é muito próxima da linguagem-objeto. Árvores são úteis durante a análise do código, quando muitas transformações do código precisam ser discutidas. Depois que a análise sintática e a análise semântica são concluídas, muitos compiladores otimizadores executam transformações adicionais que produzem uma representação intermediária mais fácil de traduzir para a máquina-objeto. A Seção 8.5 apresenta algoritmos para a geração de código intermediário para as construções típicas das linguagens de programação.

Otimização de Código

A fase de otimização de código tenta melhorar o código intermediário, de modo que seja possível obter um código-objeto mais rápido ou mais curto ou ambos. Melhorias significativas na velocidade podem ser obtidas com uma otimização global. Por exemplo, muitos algoritmos de otimização global usam técnicas de análise sofisticadas para descobrir informações sobre o fluxo de dados do programa e executar transformações com base nessas informações. Os Capítulos 9 e 10 discutem algumas dessas técnicas. Algumas otimizações podem ser aplicadas a programas-objeto após a geração de código, como veremos no Capítulo 8. Uma otimização simples pode reduzir consideravelmente o tempo de execução de um programa-objeto, mesmo com uma boa arquitetura de compilador.

Um exemplo de otimização é a eliminação de expressões comuns. Em

```
a := b + c
b := a - d
c := b + c
d := a - d                                              (1.4)
```

as duas atribuições a `d` têm o mesmo valor, pois nem `b` nem `c` são modificados entre elas. Além disso, a segunda atribuição a `c` é igual à atribuição inicial a `a`. Portanto, o código pode ser otimizado para

```
a := b + c
b := a - d
c := a
d := b                                                  (1.5)
```

Nesse exemplo, o código original é preferível, pois uma atribuição de cópia pode ser executada durante a fase de otimização de código. Outra coisa é que o computador pode ter uma instrução de subtração que é executada mais rapidamente do que uma instrução de cópia. Não há como saber sempre qual sequência é melhor sem conhecer detalhes sobre o ambiente de execução-alvo. Portanto, o compilador pode ter que gerar ambas as sequências e avaliar o custo de cada uma delas.

Existe uma grande variação na quantidade de otimização de código que os compiladores executam. Nos dias em que o tempo de compilação era importante (por exemplo, quando os computadores eram lentos), alguns compiladores evitavam completamente as otimizações. Quando o tempo de execução é mais importante do que o tempo de compilação, como é o caso de programas de produção que rodam durante muitas horas, ou quando o programa será executado em muitos computadores diferentes, o tempo de compilação adicional é tolerável. Muitas delas são discutidas no Capítulo 9, enquanto que o Capítulo 10 discute as tecnologias usadas pelos compiladores otimizadores mais poderosos.

Geração de Código

A fase final do compilador é a geração do código-objeto, que consiste normalmente em código de máquina relocável ou código de montagem. As localizações de memória são selecionadas para cada uma das variáveis usadas pelo programa. Depois, as instruções de cada uma das variáveis são escolhidas para uma sequência de instruções de máquina que realizam a mesma tarefa. Um aspecto crucial é a alocação das variáveis aos registradores.

Por exemplo, usando os registradores R1 e R2, a tradução do código de (1.3) poderia ser assim

[illegible] (1.5)

O primeiro e o segundo operandos de cada instrução especificam a fonte e o destino, respectivamente. O F em cada instrução nos informa que as instruções lidam com números em ponto flutuante. Esse código copia o conteúdo do endereço [illegible] no registrador 2 e, então, multiplica-o pela constante 60.0. O # significa que 60.0 deve ser tratado como uma constante. A terceira instrução copia o conteúdo de [illegible] no registrador 1 e o adiciona ao valor previamente computado no registrador 2. Finalmente, o valor no registrador 1 é copiado para o endereço de [illegible], de tal forma que o código implementa a atribuição na Fig. 1.10. O Capítulo 9 cobre a geração de código.

1.4 OS PRIMOS DO COMPILADOR

Como vimos na Fig. 1.3, a entrada para o compilador pode ser produzida por um ou mais pré-processadores e pode ser necessário processamento posterior da saída do compilador, antes do código de máquina ser obtido. Nesta seção, discutiremos o contexto no qual um compilador tipicamente opera.

Pré-processadores

Os pré-processadores produzem entrada para compiladores. Podem realizar as seguintes funções:

1. *Processamento de macros.* Um pré-processador pode permitir que um usuário defina macros que sejam abreviações para construções mais longas.

2. *Inclusão de arquivos.* Um pré-processador pode incluir arquivos no papel de cabeçalhos do texto do programa. Por exemplo, o pré-processador C faz com que o conteúdo do arquivo [illegible] substitua o enunciado [illegible] ao processar um arquivo contendo tal enunciado.

3. *Pré-processadores "racionais".* Tais pré-processadores expandem as linguagens mais antigas com facilidades modernas de controle de fluxo e estruturação de dados. Por exemplo, tal pré-processador poderia prover ao usuário macros embutidas para construções, tais como comandos while ou if, quando os mesmos não existissem na linguagem de programação em si.

4. *Extensores de linguagens.* Tentam conferir maior poder às linguagens, através de macros embutidas. Por exemplo, Equel (Stonebraker *et al.* [1976]) é uma linguagem de interrogação de bancos de dados embutida em C. Os enunciados começando por ## são considerados pelo pré-processador como comandos de acesso a bancos de dados, não relacionados com a linguagem C, e traduzidos em chamadas de procedimentos para rotinas que realizam tal acesso.

Os processadores de macros lidam com dois tipos de enunciados: definição e uso de macros. As definições são normalmente indicadas por algum caractere único ou palavra-chave, como [illegible] ou [illegible]. Consistem em um nome para a macro sendo definida e em um corpo, formando a definição. Frequentemente, os processadores de macros permitem parâmetros formais em suas definições, isto é, símbolos a serem substituídos por valores (um "valor" é uma cadeia de caracteres, nesse contexto). O uso de uma macro consiste na designação de uma macro através de seu nome e no fornecimento dos parâmetros atuais, isto é, valores para seus parâmetros formais. O processador de macros substitui os parâmetros formais pelos atuais no corpo da macro; por conseguinte, o corpo substituído substitui a macro em si.

Exemplo 1.2. O sistema de composição tipográfica TeX, mencionado na Seção 1.2, contém uma facilidade generalizada para macros. As definições de macros tomam a forma

[illegible] nome da macro [illegible] gabarito [illegible] corpo [illegible]

Um nome de macro é qualquer cadeia de letras precedida por uma barra invertida. O gabarito é qualquer cadeia de caracteres, com as cadeias da forma [illegible] consideradas como parâmetros formais. Esses símbolos podem também figurar no corpo, qualquer número de vezes. Por exemplo, a macro a seguir define uma citação para o *Journal of the ACM*.

[illegible]

O nome da macro é [illegible] e o gabarito é [illegible]; os parâmetros são cadeias separadas por ponto-e-vírgula, com o último seguido por um ponto. Um uso dessa macro precisa tomar a forma do gabarito, exceto que cadeias arbitrárias devem substituir os parâmetros formais.[2] Podemos, então, escrever

[illegible]

e esperar ver

J. ACM 17:4, pp. 715-728.

A parte do corpo [illegible] chama por um "J. ACM" itálico. A expressão [illegible] informa que o primeiro parâmetro atual deve ser tratado em negrito; esse parâmetro está destinado a ser o número do volume.

TeX permite que qualquer pontuação ou cadeia de textos separe o volume, número e numeração de páginas na definição da macro [illegible]. Poderíamos mesmo não ter usado pontuação alguma, caso em que TeX tomaria cada parâmetro atual como sendo ou um único caractere ou uma cadeia envolvida por { }. □

Montadores

Alguns compiladores produzem um código de montagem, como em (1.5), que é passado a um montador para processamento posterior. Outros compiladores realizam a tarefa do montador, produzindo um código de máquina relocável, que pode ser passado diretamente para um carregador/editor de ligações. Assumimos que o leitor tenha alguma familiaridade com o que uma linguagem de montagem se parece e o que um montador faça. Aqui, iremos rever o relacionamento entre o código de montagem e o código de máquina.

O *código de montagem* é uma versão mnemônica do código de máquina, na qual são usados nomes em lugar do código binário para as

---

[1] Temos evitado até aqui o importante tema da reserva de memória para os identificadores do programa-fonte. Como veremos no Capítulo 7, a organização da memória em tempo de execução depende da linguagem sendo compilada. As decisões sobre a reserva de memória são tomadas durante a geração de código intermediário ou durante a geração de código.

[2] Bem, aproximadamente [illegible]

[illegible]

[illegible]

Montagem em Duas Passagens

[illegible]

[illegible]

Carregadores e Editores de Ligação

[illegible]

[illegible]

1.5 O AGRUPAMENTO DAS FASES

[illegible]

Interfaces de Vanguarda e Retaguarda

[illegible]

de símbolos, a análise semântica e a geração do código intermediário. Uma certa quantidade de otimizações de código pode ser feita imediatamente pela interface de retaguarda. A interface de retaguarda também realiza o tratamento de erros que está associado a essas fases.

A interface de retaguarda cria um ou mais pontos de compilação que dependem da máquina alvo e que, geralmente, não dependem da linguagem fonte. Dessa forma, a linguagem intermediária da interface de retaguarda encontramos alguns aspectos das fases de otimização e de geração de código, juntamente com os aspectos de tratamento de erros e manipulação da tabela de símbolos necessários.

Tem-se tentado, durante anos, padronizar a interface de retaguarda de um compilador, criando uma interface de retaguarda associada, de forma a produzir um compilador para a nossa linguagem fonte favorita para qualquer máquina alvo. Se a interface de retaguarda tiver sido corretamente definida (basicamente, pode-se ter um problema ao representar alguns detalhes de sua interface de retaguarda, esse assunto é discutido no Capítulo 9), é bem fácil acrescentar novos compiladores para outras linguagens fonte usando uma determinada interface de retaguarda existente. Para cada nova linguagem fonte, basta escrever um interpretador de retaguarda diferente para as diferentes interfaces de retaguarda. Uma abordagem alternativa é escrever novos compiladores para a mesma máquina. Nesse caso, cria-se uma interface de retaguarda para a máquina e, então, pode-se usar qualquer um dos compiladores existentes para linguagens fonte diferentes, de forma a produzir um compilador para a mesma máquina. Novamente, existem algumas dificuldades nessa abordagem, mas muito menos do que a abordagem anterior.

Passagens

Várias fases de uma compilação são, geralmente, implementadas juntas em uma passagem, consistindo na leitura de um arquivo de entrada e da escrita em um arquivo de saída. Na prática, as fases de leitura e escrita de um compilador são agrupadas em passagens e, por conseguinte, podemos nos referir a essa discussão da compilação em termos de fases, ao invés de passagens. O Capítulo 12 discute alguns compiladores representativos e a maneira pela qual eles, na verdade, organizam as fases em passagens.

Como mencionamos, é possível que várias fases sejam agrupadas em uma única passagem e que as atividades dessas fases estejam entrelaçadas durante a execução. Por exemplo, a análise léxica, a análise sintática, a análise semântica e a geração de código intermediário poderiam ser agrupadas em uma passagem. Se assim o fizermos, o fluxo de saída após a análise léxica pode ser traduzido diretamente em código intermediário. Mais detalhadamente, podemos pensar em analisador sintático como sendo a chefia. O analisador sintático tenta descobrir a estrutura gramatical. Para isso, ele obtém tokens do analisador léxico à medida que eles são necessários, através de chamadas ao analisador léxico, a fim de que encontre a próxima token. À medida que a estrutura gramatical é descoberta, o analisador sintático chama o gerador de código intermediário para realizar a análise semântica e gerar uma parte do código. Um compilador organizado dessa forma é apresentado no Capítulo 2.

Reduzindo o Número de Passagens

É desejável ter relativamente poucas passagens, dado que torna tempo levar para ler e gravar arquivos intermediários. Por outro lado, agrupar muitas fases numa única passagem, poderemos ser forçados a manter todo o programa na memória, porque uma fase pode precisar de informações numa ordem diferente da que a fase anterior produziu. A forma interna do programa pode ser consideravelmente maior do que o programa fonte ou arquivo objeto, de forma que o espaço também pode ser um assunto importante. Porém, na maioria dos computadores modernos, esse espaço não deve ser considerado um assunto crucial.

Para algumas fases, o agrupamento em uma passagem apresenta uns poucos problemas. Por exemplo, como mencionado acima, a interface entre o analisador léxico e o sintático pode ser frequentemente limitada a um único token. Por outro lado, é frequentemente muito difícil realizar a geração de código a não ser que a representação intermediária tenha sido completamente gerada. Por exemplo, linguagens como PL/I e Algol 68 permitem que as variáveis sejam usadas antes de serem declaradas. Não podemos gerar o código alvo para uma construção se não soubermos os tipos das variáveis envolvidos na mesma. Similarmente, a maioria das linguagens permite desvios que saltam para adiante do código. Não podemos determinar o endereço desses saltos a não ser que tenhamos visto o código intermediário e o código alvo gerado para o mesmo.

Em alguns casos, é possível deixar um espaço vazio para a informação, inserir e preencher esses espaços à medida que a informação se torna disponível. Em particular, a geração do código intermediário e do código alvo podem ser frequentemente sobrepostas, após uma passagem inicial, usando uma técnica chamada de "remendo reverso". Computadores não possuem explicar todos os detalhes de que falaremos, exista a geração de código intermediário no Capítulo 8, podemos ilustrar a remendação com alguns tipos de [illegible]. Recordemos que, nessas passagens, discutimos [illegible] e duas passagens, onde na primeira criamos determinados valores identificadores que representam endereços de memória e esses endereços serão endereços à medida que forem descobertos. Uma segunda passagem substitui, então, os identificadores pelos endereços.

Podemos considerar ação das passagens como se segue. Ao se encontrar um [illegible] que seja uma [illegible] posterior, digamos

[illegible]

gerar uma [illegible] construção, com operação [illegible] para o [illegible] espaços para o endereço. Todos os instantes com espaços em branco para endereços de [illegible] são mantidos numa lista associada à entrada da tabela de símbolos para [illegible]. Os espaços em branco serão preenchidos quando finalmente encontrarmos instruções tais como

[illegible]

colocamos o valor de [illegible] no endereço especificado na instrução que o precede. Retrocedendo, então, percorremos a lista para [illegible] para todas as instruções que foram deixadas em branco pelo endereço de [illegible] os espaços em branco dos campos de endereço daquelas instruções. Essa abordagem é fácil de implementar e as instruções podem ser mantidas na memória até que todos os endereços sejam determinados.

Esse enfoque é razoável para um instrução que possua poucos campos, a serem mantidos na memória. Como as representações internas de código para uma instrução são grossas, normalmente as mesmas estão armazenadas em uma área especial, a serem acessadas sob demanda. Dessa forma, o espaço para as instruções pode ser liberado após a geração de código para a instrução. Porém, para uma instrução que possua muitos campos, de preenchimento não é muito viável. Dependendo, num compilador com um código intermediário consistente de espaços, podemos precisar de passar quando e de quem sobre a qual [illegible] ocorrer em data.

1.6 FERRAMENTAS PARA A CONSTRUÇÃO DE COMPILADORES

O desenvolvimento de um compilador, como qualquer outro programador pode usar, contamos, atualmente, com ferramentas de software. Tais como depuradores, gerenciadores de versões, customizadores e assim por diante. No Capítulo 11, veremos como algumas delas podem ser usadas para implementar um compilador. São mencionadas apenas brevemente nessa seção, são colocadas em detalhe nos capítulos apropriados.

Logo após a escrita dos primeiros compiladores, surgiram os sistemas para ajudar nesse processo. Foram frequentemente citados como *compiladores de compiladores*, *geradores de compiladores* e *sistemas de escrita de tradutores*. São praticamente orientados em torno de um modelo particular de linguagens e mais adequados para gerar compiladores de linguagens similares ao modelo.

Por exemplo, é tentador assumir que os analisadores léxicos sejam essencialmente os mesmos para todas as linguagens, exceto para

...ação, ...ssa tarefa são símbolos reconhecidos. Muitos compiladores de compiladores produzem rotinas de análise léxica úteis para este componente geral. Essas rotinas diferem somente na lista de palavras-chave reconhecidas, essa lista pode ser facilmente personalizada pelo usuário. Uma opção válida mas pouco sofisticada é ler um arquivo de reconhecedores padrão, mas isso não identifica símbolos que possam conter caracteres alfanuméricos e underscores.

Algumas ferramentas geram rotinas para o processamento de componentes específicos do compilador. Essas ferramentas usam geralmente especificações para especificar implementações e muitas usam algoritmos um tanto sofisticados. As ferramentas de maior sucesso são aquelas que escondem os detalhes do algoritmo de geração e produzem componentes que podem ser facilmente integrados à parte restante do compilador. Eis a seguir uma lista de algumas ferramentas úteis para a construção de componentes:

1. *Geradores de analisadores gerados automaticamente.* Produzem analisadores léxicos normalmente a partir de entrada baseada numa gramática livre de contexto. Nos primeiros compiladores, a análise sintática consumia, muito, grande parte do tempo de execução de um compilador típico. Também, do ponto de vista intelectual, pensava-se que a análise sintática era uma das mais difíceis de ser implementadas. Muitas das "pequenas linguagens" usadas para o projeto de linguagens de alto nível (como PIC [Kernighan 1982] e EQN foram implementadas num período de dois dias usando um gerador de *parsers* descrito na Seção 4.7. Muitos geradores de *parsers* usam algoritmos de análise gramatical que são muito caros para serem utilizados a mais.

2. *Geradores de analisadores léxicos.* Criam automaticamente analisadores léxicos, normalmente a partir de uma especificação baseada em expressões regulares, discutidas no Capítulo 3. A organização básica do analisador léxico resultante é, com efeito, um autômato finito. Um gerador de scanners típico é sua implementação são discutidos nas Seções 3.5 e 3.8.

3. *Dispositivos de tradução orientada por sintaxe.* Produzem uma coleção de rotinas que percorrem uma árvore gramatical, tal como a da Fig. 1.4, gerando código intermediário. A idéia básica é que uma ou mais "traduções" sejam associadas a cada nó da árvore gramatical e que cada tradução seja definida em termos das traduções de seus vizinhos na árvore. Tais dispositivos são discutidos no Capítulo 5.

4. *Geradores automáticos de código.* Tal ferramenta toma uma coleção de regras que define a tradução de cada operação da linguagem intermediária para a linguagem de máquina alvo. Tais regras precisam incluir detalhamentos suficientes para que possamos lidar com os diferentes métodos de acesso possíveis para os dados (por exemplo as variáveis podem estar em registradores, numa localização fixa da memória ou podem ser relativas a uma posição de uma pilha). A técnica básica é a de "casamento e substituição". Blocos grandes de código intermediário são substituídos por "gabaritos" que representam as sequências de instruções de máquina de uma forma tal que as suposições sobre armazenamento das variáveis se correspondem de gabarito a gabarito. Cuidado existe para assegurar que as relações adequadas sejam mantidas entre as variáveis que ocorrem em gabaritos vizinhos. Muitos métodos sofisticados são usados na geração de "melhor" código possível para uma dada máquina, mas o processo de geração de código mais rápido é, em princípio, simples. Uma descrição sucinta desta técnica é dada na Seção 10.11.

5. *Dispositivos de fluxo de dados.* Muito da informação necessária para se realizar uma boa otimização de código envolve a "análise de fluxo de dados", capturando-se a informação sobre como os valores são transmitidos de uma parte do programa para outra. As diferentes tarefas desta natureza podem ser realizadas essencialmente pela mesma rotina, com os usuários fornecendo detalhes de particular interesse para as computações em questão. Algoritmos de análise de fluxo de dados são discutidos no Capítulo 9.

NOTAS BIBLIOGRÁFICAS

Ao escrever em 1962, sobre a história da escrita de compiladores, Knuth [1962] observou que, "Nesse campo tem havido uma quantidade de interessantes descobertas paralelas da mesma técnica por pessoas trabalhando independentemente". Continuou, observando que vários artigos daquela época haviam de fato descoberto "vários aspectos de uma ciência que ainda não podia ser vista como existindo". Reconhecendo a natureza intelectual de muitos problemas pertinentes à tecnologia de compiladores, as notas bibliográficas serão aqui mantidas como um auxílio para estudo posterior da literatura.

As notas históricas no desenvolvimento das linguagens de programação e compiladores até a década de 60 também podem ser encontradas em Knuth e Trabb Pardo [1977], Wexelblat [1981] e em depoimentos históricos sobre várias linguagens de programação lembrados por participantes de seus desenvolvimentos.

Alguns artigos mais fundamentais na computação foram coletados em Rosen [1967] e Pollack [1972]. A edição de janeiro de 1961 do *Communications of the ACM* fornece um retrato vívido do estado da arte da escrita de compiladores daquela época. Uma descrição de todos os detalhes de um compilador Algol mais tarde foi feita por Randell e Russell [1964].

Começando no início dos anos 60, com o estudo da sintaxe, os estudos teóricos vieram a ter uma profunda influência no desenvolvimento da tecnologia de compiladores e, talvez, no mínimo, tanta influência quanto em qualquer outra área da ciência da computação. O aspecto fascinante da análise gramatical tem se desvanecido, mas a compilação como um todo continua a ser assunto de uma pesquisa viva. Os livros desta pesquisa se tornarão evidentes quando examinarmos as complicações em mais detalhes nos próximos capítulos.

Capítulo 2

UM COMPILADOR SIMPLES DE UMA PASSAGEM

Este capítulo é uma introdução ao material existente nos Capítulos 3 a 8 deste livro. Vamos criar um compilador de demonstração, escrito em Java, que traduz expressões infixas para a forma pós-fixa. Aqui, a ênfase está nos métodos da linguagem de implementação, em vez de nas análises léxica, sintática e de tradução mais refinadas, ou na geração e otimização de código intermediário. Os Capítulos 5 e 6 tratam da geração de código e da otimização.

2.1 VISÃO GERAL

Uma linguagem de programação pode ser definida pela descrição da aparência de seus programas (a sintaxe da linguagem) e do que os mesmos significam (a semântica da linguagem). Para especificar a sintaxe de uma linguagem, apresentamos uma notação amplamente usada, a gramática livre de contexto ou BNF (para Forma de Backus-Naur). Com as definições corretas, uma gramática livre de contexto pode determinar a estrutura sintática sem levar em consideração o que a estrutura significa. Mais precisamente, para especificar a sintaxe, veremos como construções numéricas, como sequências de dígitos e expressões com operadores e parênteses, podem ser descritas. Depois, veremos como a definição dirigida pela sintaxe pode ser usada para especificar a tradução de linguagens. Finalmente, veremos como um tradutor pode ser implementado. Começaremos com a construção de um programa simples para traduzir expressões infixas para expressões pós-fixas, consistindo em dígitos separados por sinais de adição e subtração. Vamos ampliar essas idéias, nos Capítulos 3 a 8, para construir um compilador para uma linguagem de programação mais ampla. Uma abordagem alternativa, que estudaremos em primeiro lugar, é a construção de um compilador para uma linguagem que não existe. Nesta seção, descrevemos duas linguagens de programação. Nas duas seções subseqüentes, mostramos como traduzir para essas linguagens. Mais adiante neste capítulo, desenvolvemos um tradutor para uma terceira linguagem. Os detalhes do desenvolvimento completo de um segundo tradutor são descritos no início do Capítulo 3.

Um tradutor dirigido pela sintaxe é construído em torno de uma definição dirigida pela sintaxe. Nesta, associamos atributos aos símbolos da gramática que descrevem a linguagem de entrada e definimos regras para calcular os valores dos atributos à medida que cada construção sintática é reconhecida. Um compilador precisa coletar informações sobre construções de uma parte do programa fonte para gerar código para outra parte. Pequenas linguagens geralmente exigem apenas uma pequena quantidade de informações sobre o programa fonte, mas mesmo assim é útil organizar as informações em relação à sintaxe, associando-as com os componentes sintáticos que as descrevem.

Para implementações simples, as definições dirigidas pela sintaxe podem ser implementadas durante a análise sintática, sem a construção explícita de uma árvore sintática ou de outra representação hierárquica. Nossa primeira implementação segue esse modelo. Ela usa a saída de um analisador sintático para registrar a estrutura sintática do programa fonte. Em seguida, usa um procedimento associado a cada produção para gerar a tradução daquela construção sintática.

Uma definição dirigida pela sintaxe especifica a tradução de construções em termos de atributos associados com seus componentes sintáticos. Um atributo pode representar qualquer coisa que queiramos associar com uma construção. Por exemplo, pode ser uma cadeia de caracteres, um número ou um tipo. Um tradutor dirigido pela sintaxe é um programa que usa uma tabela e procedimentos para percorrer uma árvore sintática e calcular regras semânticas. Durante a análise sintática ou percorrendo a árvore, sempre que uma regra é aplicada, seu procedimento associado é chamado. Esse procedimento pode gerar parte da tradução.

Em uma definição dirigida pela sintaxe, os atributos associados com cada símbolo gramatical são determinados pelas regras semânticas associadas com as produções naquele símbolo. Em uma tradução dirigida pela sintaxe, as regras semânticas são executadas à medida que as produções são aplicadas durante a análise sintática. Uma produção e suas regras semânticas descrevem como calcular os atributos associados com o cabeçalho da produção, em termos de constantes, atributos associados com os símbolos da produção e atributos associados com os símbolos não-terminais dentro da produção.

Uma sequência de tokens representando uma construção válida é chamada de sentença da linguagem. Em uma definição dirigida pela sintaxe, associamos um não-terminal com cada construção que queremos traduzir. Em geral, cada sentença da linguagem consiste em uma sequência de subconstruções, e o não-terminal associado com cada construção tem um atributo cujo valor é a tradução daquela construção. Nomeamos os atributos associados com os não-terminais e os símbolos da gramática, escrevendo o nome do atributo depois do símbolo correspondente, separados por um ponto. Suponha que E seja um não-terminal que representa uma construção válida. Se E representa a construção X op Y, então o valor de E.t é formado concatenando a tradução de X, a tradução de Y e op, nessa ordem. Se E representa a construção 9 - 5, então E.t é a cadeia de caracteres 9 5 -. Usamos a notação de concatenação em que X || Y denota a concatenação das cadeias de caracteres X e Y. Outras regras semânticas podem ser definidas para especificar como os atributos são gerados.

Uma definição dirigida pela sintaxe não impõe uma ordem específica na qual os atributos são calculados ou os nós da árvore sintática são percorridos. Qualquer ordem que calcule os atributos corretamente é aceitável. Uma definição dirigida pela sintaxe que especifica a tradução construída pela maioria das linguagens de programação é baseada em uma ordem de tradução chamada de percurso em profundidade, que começa na raiz e visita os nós da árvore sintática de uma forma genérica. Um percurso em profundidade específico que é útil para definições dirigidas pela sintaxe é mostrado na Fig. 2.1. Uma definição dirigida pela sintaxe com atributos associados com não-terminais e regras semânticas associadas com produções é chamada de definição dirigida pela sintaxe. Uma tradução dirigida pela sintaxe é uma forma de definição dirigida pela sintaxe na qual os atributos são avaliados e as regras semânticas são executadas durante a análise sintática. Em uma tradução dirigida pela sintaxe, as regras semânticas de uma produção são executadas quando a produção é aplicada durante a análise sintática.

Em vez de construir explicitamente a árvore sintática, podemos executar as ações correspondentes à medida que a árvore é construída implicitamente. Por exemplo, suponha que a última ação em uma regra semântica associada com a produção E → X op Y seja imprimir o op. Então, em um percurso em profundidade, X.t será impresso, depois a subárvore para Y será percorrida e, finalmente, op será impresso, quando o nó para E for visitado pela última vez. Em um percurso em profundidade, cada nó é visitado várias vezes durante o percurso, pois o processo precisa retornar a um nó depois de visitar cada um de seus filhos. Geralmente, os filhos são visitados da esquerda para a direita. Em um percurso em profundidade, os nós são visitados na ordem em que seus registros de ativação são criados e destruídos durante a execução de um programa que percorre a árvore. Em um percurso em profundidade, cada nó é visitado várias vezes durante o percurso, pois o processo precisa retornar a um nó depois de visitar cada um de seus filhos. Geralmente, os filhos são visitados da esquerda para a direita. Em um percurso em profundidade, os nós são visitados na ordem em que seus registros de ativação são criados e destruídos durante a execução de um programa que percorre a árvore.

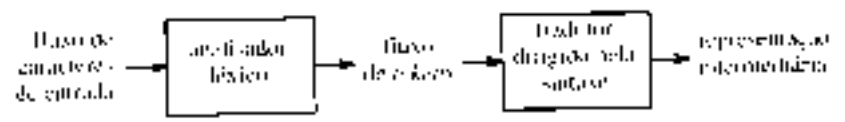


Fig. 2.1. Estrutura de um tradutor típico.

...ntes à esquerda e à direita, como "podem ser formadas". Informalmente, chamamos de uma *produção* uma dessas regras, embora mais tarde tenhamos um termo mais preciso. Símbolos em produções, como *expr*, *expr1* e *digito* na produção (2.3), são chamados de *variáveis*. Os símbolos, como os dígitos 0 e 1 em (2.3), são chamados de *terminais*.

Uma *gramática livre de contexto* possui quatro componentes:

1. Um conjunto de terminais, conhecidos como *símbolos terminais*.
2. Um conjunto de não terminais.
3. Um conjunto de produções, cada uma consistindo de um não terminal, chamado de *lado esquerdo* da produção, uma seta e uma sequência de zero ou mais terminais e não terminais, chamados de *símbolos direitos* da produção.
4. Uma designação de um dos símbolos não terminais como o *símbolo inicial*.

Seguimos a convenção de especificar gramáticas pela listagem de suas produções, com aquela para o símbolo inicial listada primeiro. Assumimos que dígitos e sinais tais como + e − são símbolos terminais. Assumimos que qualquer símbolo que não seja um dígito, um sinal ou uma seta é um não terminal, e que o lado esquerdo de uma produção contém apenas um símbolo não terminal. Finalmente, assumimos que produções com o mesmo não terminal à esquerda podem ter todos os seus lados direitos agrupados, com os lados direitos alternativos separados pelo símbolo "ou" |.

Exemplo 2.1. Vamos exemplificar neste capítulo uma expressão simples que lista ou soma dígitos, como "9-5+2". Uma gramática para essas expressões é

$$
\begin{aligned}
lista &\rightarrow lista + digito \\
lista &\rightarrow lista - digito \\
lista &\rightarrow digito \\
digito &\rightarrow 0 \mid 1 \mid \cdots \mid 9
\end{aligned}
\qquad (2.1)\ (2.2)\ (2.3)\ (2.4)
$$

Os lados direitos das três produções com o não terminal *lista* à esquerda mostram como listas podem ser construídas: a primeira produção diz que uma lista pode ser uma lista seguida por um sinal de mais e então um dígito, a segunda produção é similar, exceto que o sinal de menos é usado no lugar do sinal de mais, e a terceira produção diz que uma lista pode ser apenas um dígito. Notamos que a produção (2.4) para *digito* lista explicitamente uma produção para cada um dos dez símbolos dígitos.

De acordo com nossas convenções, os símbolos da gramática são assim classificados:

–

Os exemplos a seguir mostram como essa gramática pode ser usada para definir, *formalmente*, a sintaxe de listas de dígitos separados por sinais de mais e de menos. Começaremos a discussão de como a sintaxe pode ser especificada e como a especificação pode ser usada para organizar o restante do compilador.

Dizemos que uma produção é *para* um não terminal se o lado esquerdo da produção for esse não terminal. Uma *cadeia* de dígitos é uma sequência de um ou mais símbolos que representam um dígito. A cadeia "9-5+2" consiste de três cadeias de dígitos separadas por sinais de mais ou de menos.

Uma gramática deriva cadeias começando com o símbolo inicial e repetidamente substituindo uma cadeia pelo lado direito de uma produção para aquele não terminal. As cadeias que podem ser derivadas a partir do símbolo inicial formam a *linguagem* definida pela gramática.

Exemplo 2.2. A linguagem definida pela gramática do Exemplo 2.1 consiste em listas de dígitos separados por sinais de mais e de menos.

As dez produções para o não terminal *digito* permitem substituir qualquer um dos dígitos pela cadeia de um único dígito. Pela produção 2.2, uma lista pode ser uma lista, um sinal de mais e um dígito. As produções 2.3 e 2.4 são semelhantes, mas usam o sinal de menos. Pela produção 2.4, uma lista pode ser apenas um dígito.

Seguem algumas das maneiras pelas quais podemos mostrar que a cadeia 9-5+2 pertence à linguagem desta gramática. Por exemplo, podemos deduzir que 9-5+2 é uma lista, como segue:

a) 9 é uma lista pela produção (2.4), pois 9 é um dígito.
b) 9-5 é uma lista pela produção (2.3), pois 9 é uma lista e 5 é um dígito.
c) 9-5+2 é uma lista pela produção (2.2), pois 9-5 é uma lista e 2 é um dígito.

Essa sustentação está dos mais para os menos. Fig. 2.3 ilustra como a árvore está ligada por um símbolo da gramática. Um nó interior e seus filhos correspondem a uma produção; o nó interior corresponde ao lado esquerdo da produção, os filhos, ao lado direito. Tais árvores são chamadas de *árvores gramaticais* e são estudadas a seguir.

Exemplo 2.3. Uma especificação mais detalhada de listas de dígitos separados por sinais de mais e de menos pode incluir ações semânticas. Vamos construir um programa que traduz tais listas para expressões terminadas por sinais de mais, também chamadas de expressões em *notação pós-fixada*. Assim, a tradução de "9-5+2" é "95-2+". Cadeias que não são listas, como "52+", não têm tradução definida.

Começaremos com apenas o lado direito da primeira produção:

$$lista \rightarrow lista + digito$$

Para traduzir uma lista seguida por um sinal de mais e um dígito, traduzimos a lista, seguida pelo dígito, e então pelo sinal de mais. Não fornecemos uma tradução para o dígito ou para o sinal de mais isoladamente, portanto a tradução de qualquer coisa que não seja uma lista não está definida.

Em seguida, consideramos a produção

$$lista \rightarrow lista - digito$$

Novamente, traduzimos a lista, seguida pelo dígito, e então pelo sinal de menos. Essas duas regras são ilustradas na Fig. 2.4.

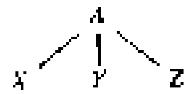
Finalmente, a tradução de um único dígito é o próprio dígito:

$$digito \rightarrow 0 \mid 1 \mid \cdots \mid 9$$

Uma árvore gramatical para 9-5+2 de acordo com a gramática do Exemplo 2.1

Fig. 2.3. Árvore gramatical para 9-5+2 de acordo com a gramática do Exemplo 2.1.

cal pode ter um nó interior rotulado A, com três filhos rotulados X, Y e Z, da esquerda para a direita:



Formalmente, dada uma gramática livre de contexto, uma árvore gramatical possui as seguintes propriedades:

1. A raiz é rotulada pelo símbolo de partida.
2. Cada folha é rotulada por um token ou por ε.
3. Cada nó interior é rotulado por um não-terminal.
4. Se A é um não-terminal rotulando algum nó interior e X, X ... X são os rótulos dos filhos daquele nó, da esquerda para a direita, então A → X X ... X é uma produção. Aqui, X, X ... X figuram no lugar de símbolos que sejam terminais ou não-terminais. Como um caso especial, se A → ε, então um nó rotulado A deve possuir um único filho rotulado ε.

Exemplo 2.4. Na Fig. 2.2, a raiz é rotulada *lista*, o símbolo de partida da gramática do Exemplo 2.1. Os filhos da raiz são rotulados, da esquerda para a direita, *lista*, + e *dígito*. Note que

lista → *lista* + *dígito*

é uma produção na gramática do Exemplo 2.1. O mesmo padrão com − é repetido para o filho à esquerda da raiz e os três nós rotulados *dígito* têm, cada um, um filho que é rotulado por um dígito. □

As folhas da árvore gramatical, lidas da esquerda para a direita, formam o produto da árvore, que é a cadeia gerada ou derivada a partir do não-terminal à raiz da árvore gramatical. Na Fig. 2.2, a cadeia gerada é [illegible]. Naquela figura, todas as folhas são mostradas no nível mais baixo. Contudo, não iremos caracterizar as folhas dessa forma. Qualquer árvore impõe uma ordem natural, da esquerda para a direita, às suas folhas, baseada na idéia de que se a e b são dois filhos com um mesmo pai e a está à esquerda de b, então todos os descendentes de a estão à esquerda dos descendentes de b.

Outra definição da linguagem gerada por uma gramática é a do conjunto de cadeias que podem ser geradas por alguma árvore gramatical. O processo de encontrar uma árvore gramatical para uma dada cadeia de tokens é chamado de *análise gramatical* ou *análise sintática* daquela cadeia.

Ambigüidade

Temos que ser cuidadosos ao falar sobre a estrutura de uma cadeia segundo uma gramática. Conquanto seja claro que cada árvore gramatical derive exatamente a cadeia formada por suas folhas, uma gramática pode ter mais de uma árvore gramatical gerando uma dada cadeia de tokens. Tal gramática é dita *ambígua*. A fim de mostrar que uma gramática é ambígua, tudo o que precisamos fazer é encontrar uma cadeia de tokens que tenha mais de uma árvore gramatical. Como uma

Fig. 2.3. Duas árvores gramaticais para [illegible]

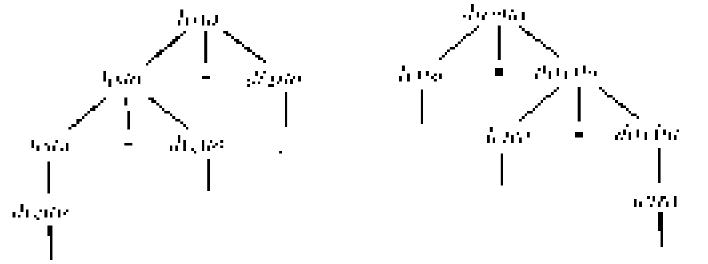


Fig. 2.4. Árvores gramaticais para operadores associativos à esquerda e à direita.

cadeia com mais de uma árvore possui usualmente mais de um significado, precisamos projetar, para as aplicações de compilação, gramáticas não ambíguas ou usar gramáticas ambíguas com regras adicionais para resolver as ambigüidades.

Exemplo 2.5. Suponha que no Exemplo 2.1 não fizéssemos distinção entre dígitos e listas. Poderíamos ter escrito a gramática

cadeia → *cadeia* + *cadeia* | *cadeia* − *cadeia* | [illegible]

Combinando-se as noções de *dígito* e *lista* no não-terminal *cadeia* faz sentido apenas superficialmente, pois um *dígito* é um caso especial de uma *lista*.

Entretanto, a Fig. 2.3 mostra que uma expressão como [illegible] + [illegible] possui agora mais de uma árvore gramatical. As duas árvores para [illegible] correspondem às duas formas de parentetizar a expressão [illegible] e [illegible]. Esta segunda forma de parentetização dá à expressão o valor 2, ao invés do valor costumeiro 6. A gramática do Exemplo 2.1 não permitia essa interpretação. □

Associatividade dos Operadores

Convencionalmente, [illegible] é equivalente a [illegible]. Quando um operando, como [illegible], possui operadores à esquerda e à direita, são necessárias convenções para decidir que operador recebe que operando. Dizemos que o operador + associa à esquerda porque um operando com sinais de adição em ambos os lados é apanhado pelo operador à sua esquerda. Na maioria das linguagens de programação, os quatro operadores aritméticos — de adição, subtração, multiplicação e divisão — são associativos à esquerda.

Alguns operadores comuns, tais como a exponenciação, são associativos à direita. Como outro exemplo, o operador de atribuição = em C é associativo à direita; em C, a expressão [illegible] é tratada da mesma forma que a expressão [illegible].

Cadeias, como [illegible], com um operador associativo à direita são geradas pela seguinte gramática:

direita → *letra* = *direita* | *letra*
letra → [illegible]

O contraste entre uma árvore gramatical para um operador associativo à esquerda, como −, e uma árvore gramatical para um operador associativo à direita, como =, é mostrado na Fig. 2.4. Note que a árvore gramatical para [illegible] cresce para baixo em direção à esquerda, enquanto que a de [illegible] cresce para baixo e à direita.

Precedência de Operadores

Considere a expressão [illegible]. Existem duas possíveis interpretações para a mesma: [illegible] ou [illegible]. A associatividade de + e de * não resolve essa ambigüidade. Por esse motivo, precisamos co-

obedece a precedência e associatividade dos operadores quando mais de um tipo pode estar presente.

Discutimos como passar propriedades de construções para outras construções e como avaliar definições dirigidas pela sintaxe. Em seguida, vamos esboçar algumas aplicações típicas das definições dirigidas pela sintaxe. Detalhes de algumas aplicações serão dados mais tarde.

2.3 TRADUÇÃO DIRIGIDA PELA SINTAXE

A tradução dirigida pela sintaxe é um método informal para descrever a tradução de construções de linguagens de programação. Ela associa informações a uma construção de linguagem de programação através de atributos. Os atributos podem representar qualquer coisa que desejarmos: um valor, um tipo, um local de memória ou uma sequência de instruções. Escrevemos uma especificação dirigida pela sintaxe para descrever a tradução de construções em termos de atributos associados a essas construções. Uma especificação dirigida pela sintaxe oculta detalhes de implementação e isenta o usuário de ter que especificar explicitamente a ordem em que a tradução acontece. Especificações dirigidas pela sintaxe ajudam a projetar tradutores e também podem ser usadas em ferramentas que constroem tradutores.

Nesta seção, definimos um formalismo, chamado de definição dirigida pela sintaxe, para especificar traduções para construções de linguagens de programação. Uma definição dirigida pela sintaxe especifica a tradução de uma construção em termos de atributos associados aos seus componentes sintáticos. Em capítulos posteriores, definições dirigidas pela sintaxe são usadas para especificar muitas das traduções que ocorrem na vanguarda de um compilador.

Introduzimos, também, uma notação mais procedimental chamada de esquema de tradução, para especificar traduções. Ao longo deste capítulo, usamos esquemas de tradução para passar expressões infixas para a notação posfixa. Uma discussão mais detalhada das definições dirigidas pela sintaxe e esquemas de tradução está contida no Capítulo 5.

Notação Posfixa

A *notação posfixa* para uma expressão E pode ser definida recursivamente como se segue:

1. Se E for uma variável ou uma constante, então a notação posfixa para E será o próprio E.
2. Se E for uma expressão da forma $E_1 \; op \; E_2$, onde op é qualquer operador binário, então a notação posfixa para E será $E_1' \; E_2' \; op$, onde E_1' e E_2' são as notações posfixas para E_1 e E_2, respectivamente.
3. Se E for uma expressão da forma (E_1), então a notação posfixa para E_1 será também a notação posfixa para E.

Os parênteses não são necessários na notação posfixa porque a posição e a aridade (número de argumentos) dos operadores permitem somente uma única decodificação de uma expressão posfixa. Por exemplo, a notação posfixa para $(9-5)+2$ é $95-2+$, e a notação posfixa para $9-(5+2)$ é $952+-$.

Definições Dirigidas pela Sintaxe

Uma *definição dirigida pela sintaxe* é uma generalização de uma gramática livre de contexto na qual a cada símbolo da gramática associa-se um conjunto de atributos, e a cada produção associa-se um conjunto de regras semânticas para computar os valores dos atributos associados aos símbolos que aparecem naquela produção. A gramática e o conjunto de regras semânticas constituem a definição dirigida pela sintaxe.

A tradução é uma especificação de saída. A saída pode ser uma cadeia de caracteres, uma sequência de instruções ou qualquer coisa que possa ser representada por uma cadeia de caracteres, uma sequência de instruções ou uma estrutura de memória arbitrária.

Suponha que uma definição dirigida pela sintaxe envolva uma gramática que tenha o símbolo da gramática X no lado esquerdo de uma produção. Cada ocorrência de X em uma árvore gramatical para essa gramática é um nó rotulado por X. Se X tiver um atributo a, então cada nó rotulado por X terá um valor a associado. Uma árvore gramatical mostrando os valores dos atributos em cada nó é chamada de *árvore gramatical anotada*.

Atributos Sintetizados

Um atributo é dito *sintetizado* se seu valor em um nó da árvore gramatical é determinado a partir dos valores dos atributos dos filhos daquele nó. Os atributos sintetizados possuem a desejável propriedade de que

Produção	Regra Semântica
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]

Fig. 2.5 [illegible]

podem ser avaliados durante uma única caminhamento em profundidade da árvore gramatical. Neste capítulo, somente atributos sintetizados serão estudados; os atributos herdados serão considerados no Capítulo 5.

Exemplo 2.6. Na Fig. 2.5 é mostrada uma definição guiada pela sintaxe para calcular expressões aritméticas envolvendo os operadores sobre dígitos inteiros de ponto flutuante. Associado com cada não-terminal está um atributo, que é usado para armazenar o valor de cada expressão representada. A função semântica para a produção associada com a primeira produção simplesmente imprime o valor da expressão gerada pela subárvore enraizada no primeiro não-terminal. Quando a produção $expr \rightarrow expr + termo$ é aplicada, o valor de $expr$ é a soma dos valores de $expr$ e $termo$.

A definição guiada pela sintaxe é uma especificação conveniente de alto nível para traduções, sendo suficiente para a maioria das aplicações que precisam ser executadas durante a análise sintática. Entretanto, algumas traduções precisam de detalhes de implementação mais explícitos. Por exemplo, a definição guiada pela sintaxe para atribuições de tipos em declarações não distingue entre a primeira identificação de uma lista de identificadores e as demais. Em geração de código, é preciso saber o tipo da primeira identificação para alocar espaço e para acessar a tabela de símbolos durante a inserção de cada identificação. Devido a essas sutilezas, um esquema de tradução, que é uma notação relacionada mais de perto com os detalhes de implementação desejados, é usado com mais freqüência para especificar traduções.

Esquemas de tradução são discutidos na próxima seção. Atributos herdados são revistos na Seção 2.8, onde são usados para verificar se o uso de identificadores é consistente com suas declarações. Atributos herdados também são usados na Seção 2.8 para construir uma árvore sintática para uma expressão. Mais exemplos aparecem no Capítulo 5.

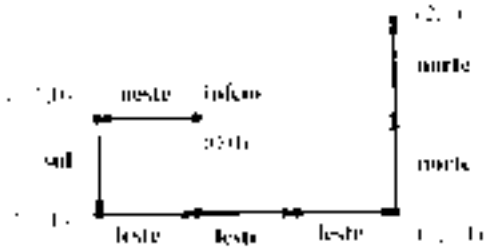


Fig. 2.7. [illegible]

de suas posições. A lista de instruções é gerada pela seguinte sequência de passos:

[illegible]	**início**
[illegible]	**leste** **norte** **oeste** **sul**

As instruções na posição dois e três são geradas

início oeste sul leste leste leste norte norte

como mostrado na Fig. 2.7.

Na figura, a cada posição marcada por um parêntese reto está representado o número de passos para o leste, norte, respectivamente, quando a posição atual é o início. Se for negativo, então o módulo da posição é a posição anterior, e a instrução é oeste ou sul, então o valor atual da posição é zero.

A tradução é executada pela sintaxe para traduções em duas sequências de instruções na posição dois e três. Cada um dos atributos, $oeste$ e $norte$, é utilizado para manter a posição atual em relação à posição de início. A sequência de instruções é gerada pela regra semântica associada com a produção $lista \rightarrow \mathbf{início}$. Como $oeste = 0$ e $norte = 0$ supondo que na posição dois e três são colocadas instruções, a sequência é inicialmente vazia. A função $gera$ concatena seus argumentos, que são cadeias, juntamente com uma nova instrução, e retorna a cadeia resultante. A regra semântica associada com a produção $lista \rightarrow lista_1\ instrução$ atualiza $oeste$ e $norte$ de acordo com a instrução gerada.

A atualização da posição de início é uma instrução condicional, dada pelas regras semânticas associadas com $instrução \rightarrow \mathbf{leste}$. Se $oeste \geq 0$, então $oeste = 0$, supondo que na posição dois e três são colocadas instruções. Caso contrário, $oeste$ é incrementado. A instrução $gera$ é então chamada para gerar uma instrução **leste**. A função de geração de código, dada pelas regras

$$instrução \rightarrow \mathbf{norte} \quad \{ instrução.gera = gera(\text{'norte'}) \}$$
$$instrução \rightarrow \mathbf{sul} \quad \{ instrução.gera = gera(\text{'sul'}) \}$$

Uma definição dirigida pela sintaxe para traduzir uma sequência de instruções norte-oeste-sul-leste é mostrada na Fig. 2.6.

Caminhamento em Profundidade

Uma definição dirigida pela sintaxe não impõe qualquer ordem específica para a avaliação dos atributos de uma árvore gramatical. Qualquer

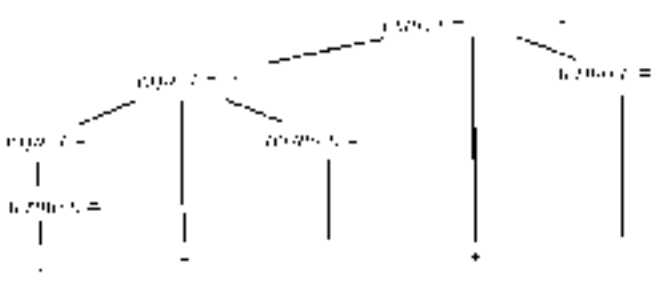


Fig. 2.8. Valores de atributos nos nós de uma árvore gramatical.

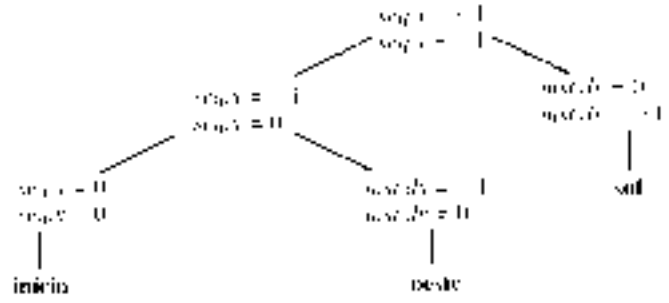


Fig. 2.9. Árvore gramatical [illegible] início oeste sul.

PRODUÇÃO	REGRAS SEMÂNTICAS
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	[illegible]

Fig. 2.9. [illegible]

que ordem de avaliação que computa o atributo após todos os atributos de que ele depende é evidente. De modo geral, durante a construção da árvore gramatical, algum atributo de um nó deve ser avaliado quando sua computação é pedida pela primeira vez, momento em que todos os seus filhos devem ser construídos, e pode depender dos atributos desses filhos daquele nó. As ordens de avaliação mais apropriadas serão identificadas em mais detalhes no Capítulo 5.

As traduções neste capítulo podem ser todas implementadas pela avaliação das regras semânticas para os atributos numa árvore gramatical numa ordem predeterminada. Uma *implementação* de uma árvore sintática é uma estrutura de dados ou uma sequência de [illegible]. Neste capítulo as regras semânticas serão avaliadas usando-se o esquema mais profundidade definido na Fig. 2.10. A ordem na qual os nós são visitados corresponde ao fato de que os filhos de cada nó aparecem da esquerda para a direita como mostrado na Fig. 2.11. As regras semânticas associadas aos nós são avaliadas na vez que todos os descendentes do mesmo nó tenham sido avaliados. (Lembre-se de "caminhamento em profundidade" por [illegible] visita um filho mais à esquerda ainda não visitado e não possui mais atributos a serem avaliados. Isto é, visita os nós tão distantes da raiz quanto possível, mas não além disso.)

Esquemas de Tradução

Nesta seção, apresentamos uma especificação procedimental para definir uma tradução. Um esquema de tradução é uma gramática livre de contexto na qual fragmentos de programas, chamados de ações semânticas, são associados aos lados direitos das produções. Um esquema de tradução é como uma definição dirigida pela sintaxe, exceto que a ordem de avaliação das regras semânticas é explicitamente mostrada. A posição na qual uma ação deve ser executada é mostrada envolvendo-a entre chaves e escrevendo-a no lado direito da produção, como em

[illegible]

```
procedimento [illegible]
início
    para cada filho m de B, da esquerda para a direita faça
        [illegible]
    avaliar as regras semânticas associadas a B
fim
```

Fig. 2.10. [illegible]

Fig. 2.11. [illegible]

Um esquema de tradução gera uma saída para cada sentença que tenha sido gerada pela gramática subjacente, normalmente através da execução das ações na ordem em que figurem durante o caminhamento em profundidade de uma árvore gramatical para a sentença. Por exemplo, considere uma árvore gramatical com um nó rotulado [illegible] para a produção [illegible]. A ação [illegible] será executada depois que todos os nós para [illegible] tenham sido percorridos, mas antes que o filho para [illegible] seja visitado.

Ao descrevermos uma árvore gramatical para um esquema de tradução, indicamos uma ação semântica [illegible] para a enésima filha de um nó, conectado por uma linha pontilhada ao nó de sua produção. Por exemplo, a parte da árvore gramatical para a produção e ação [illegible] desenhada como na Fig. 2.12. É um nó para uma ação semântica não possui filhos. Uma ação [illegible] da [illegible] projetado [illegible].

Emitindo Uma Tradução

Nesta seção, as ações semânticas nos esquemas de tradução [illegible] a saída de uma tradução [illegible]. Um atributo [illegible] é [illegible] cada vez. Para exemplo, [illegible] imprimindo cada [illegible] [illegible] somente uma vez, sem usar qualquer armazenamento para a tradução de subexpressões. Quando a saída de cada ação depende apenas dessa ação e [illegible] os caracteres são impressos é importante.

Note que as definições dirigidas pela sintaxe [illegible] têm a seguinte importante propriedade: a cadeia [illegible] anexada à tradução de um não-terminal no lado esquerdo de cada produção é a concatenação das traduções dos não-terminais à direita, na mesma ordem que na produção, com algumas cadeias opcionais [illegible] intercaladas. Uma definição dirigida pela sintaxe com esta propriedade é denominada *simples*. Por exemplo, considere a primeira produção e regra semântica da definição dirigida pela sintaxe na Fig. 2.5.

PRODUÇÃO	REGRA SEMÂNTICA
[illegible]	[illegible]

Aqui, a tradução de [illegible] é a concatenação das traduções de [illegible], seguida pelo símbolo [illegible]. Note que [illegible] antes de [illegible] no lado direito da produção.

Uma cadeia adicional aparece entre [illegible] na regra semântica

PRODUÇÃO	REGRA SEMÂNTICA
[illegible]	[illegible]

mais de uma cadeia pode aparecer entre as traduções de não-terminais do lado direito.

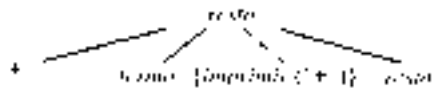


Fig. 2.12. [illegible]

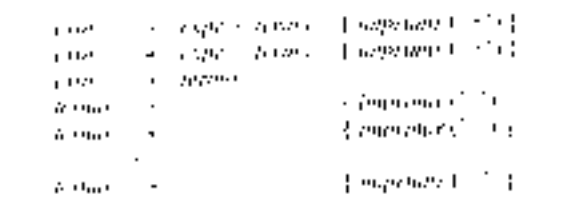


Fig. 2.13. Novo tradutor de expressões com ações pós-fixas.

Definições simples são rígidas pela sintaxe podem ser ampliadas através de esquemas de tradução nos quais ações semânticas são adicionadas às produções que ajudam na definição. As ações nas seguintes produções imprimem as cadeias dos termos em (2.6) e (2.7), respectivamente:

$$\begin{aligned} expr &\rightarrow expr_1 + termo \quad \{imprime('+')\} \\ termo &\rightarrow 0 \mid 1 \mid \ldots \mid 9 \quad \{imprime(0)\} \mid \ldots \mid \{imprime(9)\} \end{aligned}$$

Exemplo 2.4. A Fig. 2.5 contém um esquema de tradução simples para traduzir expressões na notação pós-fixa. Um esquema de tradução é uma definição dirigida pela sintaxe na qual cada símbolo da gramática tem um atributo associado. Uma árvore gramatical com anotações para o esquema é mostrada na Fig. 2.14. Note que apesar das Figs. 2.6 e 2.14 representarem o mesmo reconhecimento de entrada e saída, a tradução, nos dois casos é consideravelmente diferente. A Fig. 2.6 produz a saída à raiz da árvore gramatical, enquanto que na Fig. 2.14 a saída é impressa incrementalmente.

A raiz da Fig. 2.14 representa a primeira produção da Fig. 2.13. Num caminhamento em profundidade, nenhuma ação é executada até que o operando esquerdo expr, quando caminhamos da subárvore mais à esquerda da raiz. Visitamos, então, a folha +, na qual não existe ação. Em seguida realizamos as ações na subárvore para o operando direito termo. Finalmente a ação semântica $\{imprime('+')\}$ é executada.

Cópias das produções para fornecer termos somente no lado direito da seta de cada regra é comum em esquemas para as produções. Nenhuma cópia de expr é necessária para a produção expr → termo, somente o operador necessário — impressão para as duas produções. Quando executada durante um caminhamento em profundidade da árvore gramatical as ações da Fig. 2.14 imprimem 9 5 −.

Como uma regra geral, a maioria dos métodos de análise gramatical processa suas entradas da esquerda para a direita, ou seja, "top-down", ou da direita para a esquerda, "bottom-up". Esses métodos constroem a árvore gramatical para a entrada, ou pelo menos a sequência de regras que seriam usadas para construir a árvore, de cima para baixo (começando pela raiz) ou de baixo para cima (começando pelas folhas). Métodos mais gerais, como os usados para as gramáticas de Tipo 0 e Tipo 1 (ver Seção 2.6.1), podem ser muito ineficientes e são raramente usados na prática. Para uma classe importante de gramáticas, podemos construir analisadores eficientes, que consomem a entrada da esquerda para a direita, um símbolo de cada vez.

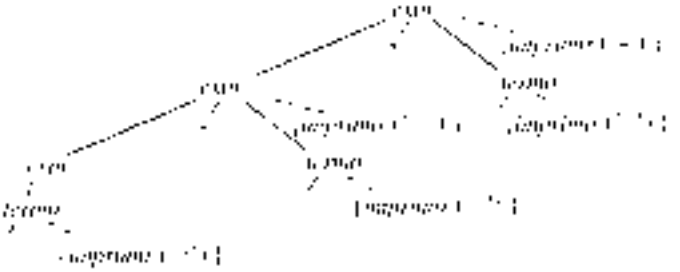


Fig. 2.14. Árvore gramatical com ações semânticas.

Durante a análise gramatical, a árvore gramatical é construída implicitamente, a menos que ações sejam inseridas para construir explicitamente a árvore. A árvore gramatical pode ser usada para guiar a geração de código subsequente. Em um compilador típico, a árvore gramatical é construída explicitamente, e a geração de código subsequente é guiada pela árvore, como discutido na Seção 2.5.

Uma análise gramatical pode ser construída para qualquer gramática livre de contexto. Algoritmos sofisticados são necessários, entretanto, para gramáticas gerais, mas os métodos desenvolvidos podem ser muito ineficientes para serem usados em produção. Felizmente, para muitas linguagens de programação úteis, podemos facilmente construir gramáticas eficientes. Algoritmos lineares bons existem para analisar essas gramáticas que surgem na prática. Analisadores gerados de linguagens de programação quase sempre realizam um único escaneamento da esquerda para a direita sobre a entrada, possivelmente com um token de cada vez.

As duas classes de métodos de análise gramatical com as quais lidaremos são chamadas de *top-down* e *bottom-up*. Esses termos se referem à ordem na qual os nós da árvore gramatical são construídos. No primeiro, a construção se inicia na raiz e prossegue em direção às folhas, enquanto que no último, a construção se inicia nas folhas e procede em direção à raiz. A popularidade dos analisadores gramaticais *top-down* é devida em parte ao fato de que analisadores eficientes podem ser construídos mais facilmente manualmente usando métodos *top-down*. Além disso, métodos *bottom-up* são, muitas vezes, capazes de manipular uma classe mais ampla de gramáticas e esquemas de tradução, então as ferramentas de construção de analisadores gramaticais diretamente aplicáveis são geralmente baseadas em métodos *bottom-up*.

A Análise Gramatical *Top-Down*

Introduzimos a análise gramatical *top-down* considerando uma gramática que é boa para esta classe de métodos. Posteriormente, nesta seção, consideraremos a construção de analisadores gramaticais *top-down* em geral. A seguinte gramática gera um subconjunto dos tipos da Pascal. Usamos o token **pontoponto** para representar a sequência de caracteres .. (duas vezes) como uma unidade.

```
tipo      → tipo-simples
          | ↑id
          | array [ tipo-simples ] of tipo
tipo-simples → integer                                    (2.8)
          | char
          | num pontoponto num
```

A construção top-down de uma árvore gramatical é feita a partir da análise pela raiz, rotulada pelo símbolo inicial de partida e realizando-se repetidamente os dois seguintes passos (veja a Fig. 2.15 para um exemplo).

1. A um nó rotulado por um não-terminal A, selecione uma das produções para A e construa filhos de acordo com os símbolos do lado direito da produção.
2. Encontre o próximo nó para o qual uma subárvore deve ser construída.

Para alguns métodos, o segundo passo é feito mais fácilmente se a árvore gramatical é construída em ordem de profundidade, ou seja, a ordem na qual os nós são alcançados quando um caminhamento em profundidade é executado. Em um caminhamento em profundidade, o nó mais recentemente visitado com filhos não-visitados é o próximo nó a ser visitado. Os filhos de um nó são visitados da esquerda para a direita.

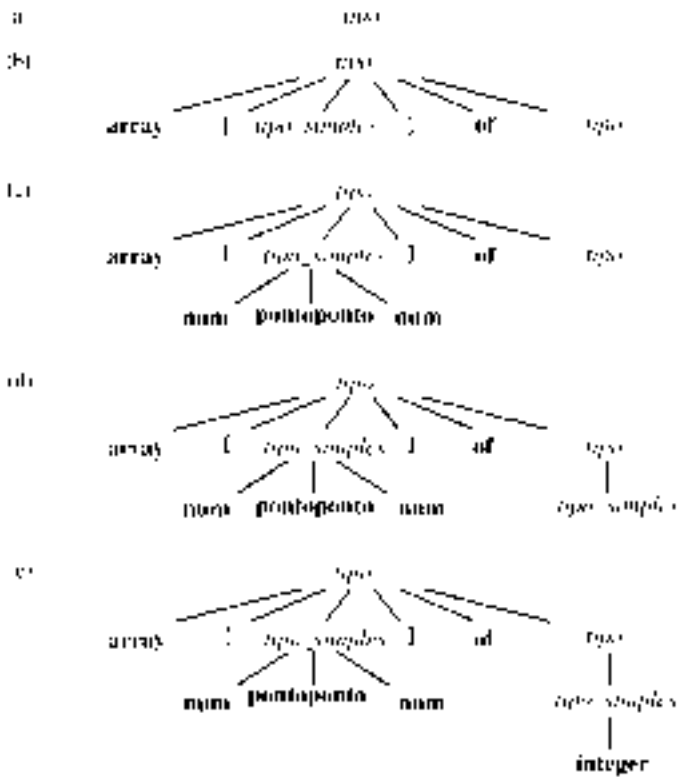


Fig. 2.15. Passos na construção sequencial de uma árvore gramatical.

Para algumas gramáticas, os passos acima podem ser implementados durante um único escaneamento da cadeia de entrada da esquerda para a direita. O texto contendo a cadeia de entrada é frequentemente referenciado como o *fluxo de entrada*. Inicialmente o símbolo corrente é o primeiro token à esquerda da cadeia de entrada. A Fig. 2.16 ilustra a análise gramatical descendente.

array [num ponto ponto num] of integer

Inicialmente, o token **array** é o símbolo corrente. A parte conhecida da árvore gramatical consiste na raiz rotulada pelo não-terminal de partida *tipo*, Fig. 2.16(a). O objetivo é construir o resto da árvore gramatical de tal forma que ela venha a gerar a cadeia de entrada.

Para oferecer uma correspondência, o não-terminal *tipo* na Fig. 2.16(a) tem seu único filho rotulado com o símbolo terminal **array**. Na gramática 2.8, existe somente uma produção para *tipo* que possa identificar uma cadeia começando com **array**. Os nós correspondentes aos filhos da raiz rotulados com os símbolos do lado direito da produção

Cada uma das árvores gramaticais na Fig. 2.16 possui setas marcando o símbolo *corrente* e o nó da árvore gramatical em consideração. Quando as árvores são construídas, os filhos para um nó são considerados em seqüência, filhos mais à esquerda. Na Fig. 2.16(b), os filhos da raiz acabaram de ser construídos e o filho mais à esquerda, rotulado por **array**, está em cheque.

Quando o nó em consideração na árvore gramatical é o de um terminal, e este rótulo corresponde ao símbolo *corrente*, como é, mas então, também na árvore gramatical quanto na entrada. O próximo *token* a entrada se torna o novo símbolo *corrente* e o próximo filho da árvore gramatical é considerado. Na Fig. 2.16(c), a seta na árvore gramatical avançou para o próximo filho da raiz e a seta à entrada avançou para o próximo *token* [. Após o avanço seguinte, a seta da árvore gramatical foi apontar para o filho rotulado pelo não-terminal *tipo_simples*. Quando um não-terminal é encontrado, uma escolha de produção é considerada necessária. O processo de selecionar uma produção para o não-terminal

Em geral, a seleção de uma produção para um não-terminal pode envolver tentativa e erro, ou seja, podemos ter que fazer uma escolha, retroceder e tentar outra produção, se a primeira escolha mostrar inadequada. Uma produção é inadequada se após substituída, um problema

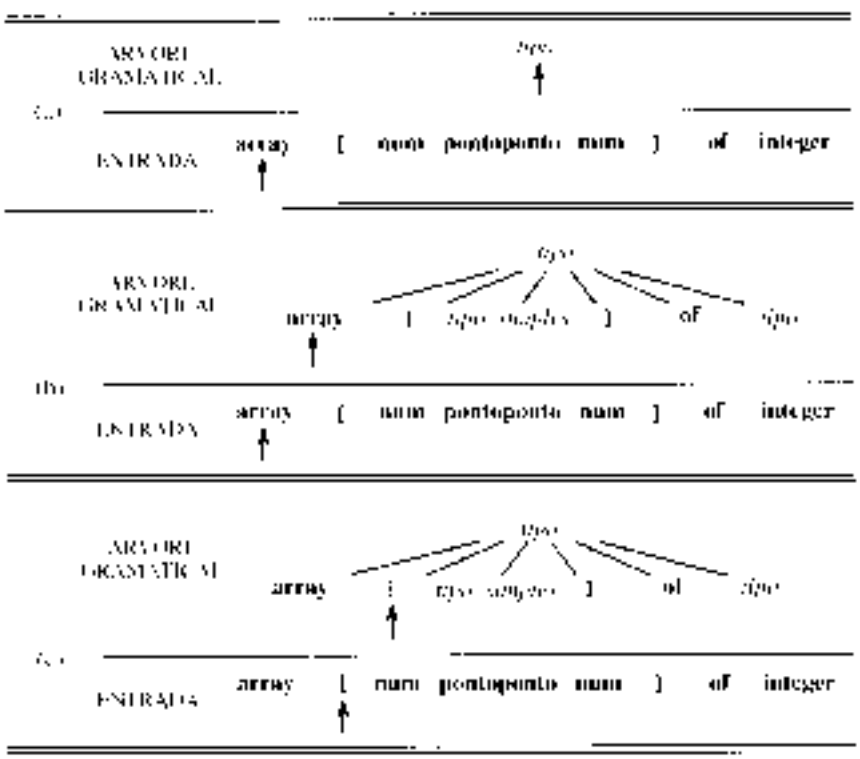

ÁRVORE GRAMATICAL
tipo
ENTRADA
array [num ponto ponto num] of integer
ÁRVORE GRAMATICAL
array [tipo_simples] of tipo
ENTRADA
array [num ponto ponto num] of integer
ÁRVORE GRAMATICAL
array [tipo_simples] of tipo
ENTRADA
array [num ponto ponto num] of integer

Fig. 2.16. Análise gramatical top-down sempre avançando a cadeia de entrada da esquerda para a direita.

compatível com a que corresponda à cadeia de entrada. Existe, no entanto, um caso especial importante: o fracasso de análise gramatical preditiva no qual o retrocesso não ocorre.

Análise Gramatical Preditiva

A análise gramatical preditiva recursiva descendente associa um procedimento *reconhece* de análise sintática, no qual executamos um conjunto de procedimentos semelhantes para processar a entrada. Um procedimento é associado a cada não terminal de uma gramática. Aqui, consideramos uma forma especial de análise gramatical descendente recursiva, chamada de análise gramatical preditiva, na qual o símbolo *lookahead* determina exatamente qual procedimento selecionado para cada não terminal. A sequência de procedimentos chamada no processamento da entrada define implicitamente uma árvore gramatical para a entrada.

O analisador gramatical preditivo da Fig. 2.17 consiste em procedimentos para os não terminais *tipo* e *tipo_simples* da gramática 2.8 e um procedimento adicional. Usamos *reconhece* a fim de simplificar o código para *tipo* e *tipo_simples*. *reconhece* avança para o próximo token de entrada se seu argumento coincidir com o símbolo *lookahead*. Do contrário, *reconhece* chama *erro*, que exibe uma mensagem de erro. Após a produção 2.8 ter sido escolhida, o código

```
reconhece (array); reconhece ('['); tipo_simples;
reconhece (']'); reconhece (of); tipo                    (2.9)
```

corresponderá ao lado direito da produção

$$tipo \rightarrow \textbf{array}\ [\ tipo\_simples\ ]\ \textbf{of}\ tipo$$

```
procedimento tipo;
início
    se lookahead = (então
        reconhece ('('); tipo_simples; reconhece (')')
    senão se lookahead = ↑ então início
        reconhece (↑); reconhece (id)
    fim
    senão se lookahead = array então início
        reconhece (array); reconhece ('['); tipo_simples;
        reconhece (']'); reconhece (of); tipo
    fim
    senão erro
fim;
procedimento tipo_simples;
início
    se lookahead = integer
        então reconhece (integer)
    senão se lookahead = char então
        reconhece (char)
    senão se lookahead = num então início
        reconhece (num); reconhece (.); reconhece (.); reconhece (num)
    fim
    senão erro
fim;
```

Fig. 2.17. Pseudocódigo para um analisador gramatical preditivo.

Note que cada terminal do lado direito é igualado com o símbolo *lookahead* e que cada não terminal do lado direito leva a uma chamada de seu procedimento correspondente.

Com a entrada da Fig. 2.16, após a leitura de **array** e [terem sido reconhecidos, o símbolo *lookahead* é num. Neste ponto o procedimento *tipo_simples* é chamado e, dentro dele mesmo, é executado o código

```
reconhece (num); reconhece (.); reconhece (.); reconhece (num)
```

O símbolo *lookahead* guia a seleção da produção a ser usada. Se o lado direito de uma produção se inicia pelo símbolo terminal atual, essa produção pode ser escolhida quando o símbolo *lookahead* for igual a esse terminal. O código acima do lado direito começando por não terminal, como em

$$tipo \rightarrow tipo\_simples \qquad (2.10)$$

pode ser escolhido de duas maneiras. Por exemplo, durante a execução do fragmento de código 2.9, sabemos que, quando o controle atingir a chamada de procedimento *tipo*, o símbolo *lookahead* será igual a **integer**. Não existe produção para *tipo* que comece com o token **integer**. No entanto, uma produção para *tipo_simples* começa por **integer**, e fazemos a produção 2.10 ser usada levando *tipo_simples* para o símbolo *lookahead* **integer**.

A análise gramatical preditiva depende da informação sobre os primeiros símbolos que podem ser gerados pelo lado direito de uma produção. Mais precisamente, seja α o lado direito de uma produção para um não-terminal A. Definimos PRIMEIRO(α) como o conjunto de tokens que figuram como os primeiros símbolos de uma ou mais cadeias geradas a partir de α. Se α é ε ou pode gerar ε, então ε também pertence a PRIMEIRO(α). Por exemplo,

$$\text{PRIMEIRO}(tipo\_simples) = \{\ \textbf{integer}, \textbf{char}, \textbf{num}\ \}$$
$$\text{PRIMEIRO}(\uparrow id) = \{\ \uparrow\ \}$$
$$\text{PRIMEIRO}(\textbf{array}\ [\ tipo\_simples\ ]\ \textbf{of}\ tipo) = \{\ \textbf{array}\ \}$$

Na prática, muitos lados direitos de produções começam por tokens, simplificando a construção dos conjuntos PRIMEIRO. O algoritmo para computar os conjuntos PRIMEIRO é dado na Seção 4.4.

Os conjuntos PRIMEIRO devem ser considerados se você estiver duas produções $A \rightarrow \alpha$ e $A \rightarrow \beta$. A análise gramatical descendente recursiva sem retrocesso requer que PRIMEIRO(α) e PRIMEIRO(β) sejam disjuntos. O símbolo *lookahead* pode então ser usado para decidir a produção a usar: se o token estiver em PRIMEIRO(α), então α é usado. Do contrário, se o símbolo *lookahead* estiver em PRIMEIRO(β), então β é usado.

Quando Usar Produções-ε

As produções com ε no lado direito requerem tratamento especial. O analisador gramatical deve, evidentemente, escolher uma produção e como um default, quando nenhuma outra puder ser usada. Como exemplo, considere

$$stmt \rightarrow \textbf{begin}\ opt\_stmts\ \textbf{end}$$
$$opt\_stmts \rightarrow stmt\ ;\ opt\_stmts \mid \varepsilon$$

Enquanto analisarmos gramaticalmente *opt_stmts*, se o símbolo *lookahead* não estiver em PRIMEIRO(*stmt*; *opt_stmts*), a produção ε é escolhida. A escolha estará certamente correta se o símbolo *lookahead* for

lexend. Qualquer outro atributo *lexdaval* que um end, no resultar numa referência descendente a *expr*.

Projetando um Analisador Gramatical Preditivo

Um *analisador gramatical preditivo* é um programa que consiste de um procedimento para cada não-terminal. Cada procedimento realiza duas tarefas.

1. Decide que produção usar através do exame do símbolo *lookahead*. A produção com lado direito que inicia com o símbolo terminal *lookahead* é escolhida se PRIMEIRO(lado direito) contém *lookahead*. Se existir um conflito entre dois lados direitos para qualquer símbolo *lookahead*, um analisador preditivo não pode ser construído com esta gramática. Uma produção com ε no lado direito é usada se o símbolo *lookahead* não estiver no conjunto PRIMEIRO para qualquer outro lado direito.

2. O procedimento casa uma produção mirando o lado direito. Um não-terminal resulta numa chamada ao procedimento para o não-terminal e um token sendo igualado ao símbolo *lookahead* resulta na leitura do próximo token de entrada. Se em algum ponto o token da produção não casa o símbolo *lookahead*, um erro é declarado. A Fig. 2.17 é o resultado da aplicação dessas regras à gramática (2.6).

Esta maneira de construir um esquema de tradução que é formado expandindo uma gramática com ações também pode ser implementada em analisadores preditivos gerados por programas como o YACC. Um algoritmo com esse propósito é dado na Seção 5.5. Para o presente, a seguinte restrição gramatical auxilia muito a construção de esquemas de tradução preditivos.

Uma gramática deve ser fatorada à esquerda se para cada não-terminal A todas as produções $A \to \alpha$ têm $\alpha = \varepsilon$ ou o primeiro símbolo de α não é o mesmo.

Quando $\alpha \neq \varepsilon$, α e ε não podem ambos derivar cadeias começando com o mesmo terminal.

Se uma gramática não é fatorada à esquerda, ela pode ser fatorada à esquerda através de uma construção simples. Para cada não-terminal A com duas produções $A \to \alpha\beta_1 \mid \alpha\beta_2$ onde $\alpha \neq \varepsilon$, substitua todas as produções A por

$$A \to \alpha A' \mid \gamma_1 \mid \cdots \mid \gamma_n$$

onde A' é um novo não-terminal e $\gamma_1 \mid \cdots \mid \gamma_n$ são todas as produções A cujo lado direito não começa com α. Finalmente substitua as produções A' por $\beta_1 \mid \cdots \mid \beta_n$. Por exemplo, com

$$\begin{aligned} stmt &\to \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ &\mid \textbf{if } expr \textbf{ then } stmt \\ &\mid a \end{aligned}$$

$expr \to b$

o não-terminal A é fatorado à esquerda porque a produção $A \to \alpha\beta_1 \mid \alpha\beta_2$ possui $\alpha = \varepsilon$ para o símbolo não-terminal $stmt$. Após a substituição das produções A como sugerido acima, a gramática resultante é

$$\begin{aligned} stmt &\to \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \; A' \\ &\mid a \\ A' &\to \textbf{else } stmt \mid \varepsilon \\ expr &\to b \end{aligned}$$

Agora, A' é o novo não-terminal. Após a fatoração à esquerda, uma gramática pode ser processada por um analisador preditivo.

Finalmente, considere o não-terminal $stmt$ e duas de suas produções

$$stmt \to \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt$$
$$\mid \textbf{if } expr \textbf{ then } stmt$$

Em muitas linguagens de programação, incluindo C, Java e Pascal, a palavra reservada **else** é casada com a palavra reservada **then** anterior mais próxima. Assim, **else** E_2 em

$$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2$$

é casada com **then** E_2. Isto é, a gramática é ambígua.

Uma regra de desambiguação que diz "case **else** com o **then** anterior mais próximo" pode ser incorporada diretamente em um analisador preditivo. Ou seja, quando A' é expandido, a ação preferida é aquela que mantém o **else** na pilha, adiando qualquer decisão sobre A' até que mais entrada seja vista. Em geral, ambiguidades podem ser tratadas em analisadores preditivos escolhendo-se a primeira produção que se aplica em uma situação. Se a ambiguidade for tratada de qualquer outra forma, como uma regra de precedência para operadores, ela pode frequentemente ser reescrita para se adequar a um analisador preditivo.

Recursão à Esquerda

É possível que um analisador gramatical descendente recursivo entre em um laço infinito. O problema ocorre em produções recursivas à esquerda, tais como

$$expr \to expr + term$$

quando o símbolo da região à esquerda do lado direito é o mesmo que o não-terminal no lado esquerdo da produção. Suponha que o procedimento para $expr$ decida aplicar esta produção. O lado direito da produção começa com $expr$, de tal forma que o procedimento para $expr$ será chamado recursivamente, e o analisador não terá progredido. Note que o símbolo *lookahead* muda somente quando um terminal do lado direito é reconhecido. Porque a produção começa pelo não-terminal $expr$, nenhuma mudança foi feita ao lado direito imediatamente antes de uma chamada recursiva, com o qual um laço infinito ocorre.

Uma produção recursiva à esquerda pode ser eliminada pela reescrita da produção ofensiva. Considere um não-terminal A com duas produções

$$A \to A\alpha \mid \beta$$

onde α e β são sequências de terminais e não-terminais que não começam por A. Por exemplo, em

$$\begin{aligned} expr &\to expr + term \\ A &= expr \quad \alpha = + term \quad \beta = term. \end{aligned}$$

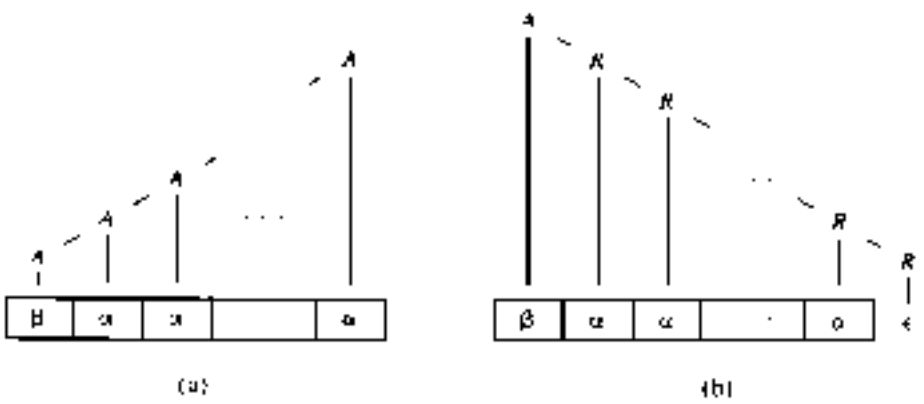
O não-terminal A é recursivo à esquerda porque a produção $A \to A\alpha$ possui o próprio A como seu símbolo mais à esquerda. A aplicação repetida dessa produção constrói uma sequência de α's à direita de A, como na Fig. 2.18(a). Quando A finalmente é substituído por β, temos β seguido por uma sequência de zero ou mais α's.

O mesmo efeito pode ser alcançado, como na Fig. 2.18(b), reescrevendo as produções para A da seguinte maneira

$$\begin{aligned} A &\to \beta R \\ R &\to \alpha R \mid \varepsilon \end{aligned} \tag{2.11}$$

Aqui, R é um novo não-terminal. A produção $R \to \alpha R$ é recursiva à direita porque a produção para R possui o próprio R como último símbolo no lado direito. Produções recursivas à direita levam a árvores que crescem para baixo em direção à direita, como na Fig. 2.19(b). As árvores que crescem para baixo em direção à direita tornam mais difícil traduzir expressões contendo operadores associativos à esquerda, como o sinal de menos. Na próxima seção, entretanto, veremos que a tradução apropriada de expressões na notação pós-fixa pode ainda ser alcançada por um projeto simultâneo de esquema de tradução baseado numa gramática recursiva à direita.

No Capítulo 4, consideraremos formas mais gerais de recursão à esquerda e mostraremos como toda recursão à esquerda pode ser eliminada de uma gramática.



(a) (b)

Fig. 2.18. Formas recursivas à esquerda e à direita para uma gramática.

2.5 UM TRADUTOR PARA EXPRESSÕES SIMPLES

Usando as técnicas das três últimas seções, construiremos agora um tradutor dirigido pela sintaxe, sob a forma de um programa C em uma

```
expr   → expr + termo   { imprima('+') }
       | expr - termo   { imprima('-') }
       | termo
termo  → 0               { imprima('0') }
       | 1               { imprima('1') }
       ...
       | 9               { imprima('9') }
```

Fig. 2.19: Esquema gramatical de uma tradução da forma infixa para a posfixa.

dez expressões aritméticas para a forma infixa. A fim de manter o programa inicial grandemente propício, começamos por expressões constituídas de dígitos separados por sinais de mais e de menos. A linguagem é estendida nas duas direções seguintes de forma a incluir números, identificadores e outros operadores. Dado que as expressões ligam-se em pontos construção-a-construção de linguagens, é útil estudarmos sua tradução em detalhes.

Uma esquema de tradução dirigida pela sintaxe pode frequentemente servir como especificação para um tradutor. O esquema da Fig. 2.19 (repetido da Fig. 2.1) executa a definição da tradução ao ser realizada. (Como de costume, o esquema é baseado na gramática sintática de um dado esquema precisa ser modificada antes que possa ser analisada eficazmente por um analisador preditivo. Em particular, a gramática subjacente ao esquema da Fig. 2.19 é recursiva à esquerda, como vimos na última seção, um analisador preditivo não pode tratar uma gramática recursiva à esquerda. Para eliminar a recursão à esquerda, podemos obter uma gramática adequada para o tradutor preditivo descendente recursivo.

Sintaxe Abstrata e Concreta

Um ponto de partida útil para se pensar sobre a tradução de uma cadeia de entrada é a construção de uma árvore sintática, na qual cada nó representa um operador e os filhos de um nó representam os operandos. Por comodidade uma árvore gramatical é chamada uma *árvore sintática* para a gramática subjacente com a qual ela está associada. Árvores sintáticas abstratas, ou simplesmente árvores sintáticas, diferem das árvores gramaticais pela ausência de variáveis não terminais de forma, descompactando para a tradução, não digo nada nas árvores sintáticas.

Por exemplo, a árvore sintática para `9-5+2` é mostrada na Fig. 2.20. Como `+` e `-` possuem o mesmo nível de precedência e os operadores de mesmo nível de precedência são avaliados da esquerda para a direita, a árvore mostra `9-5` agrupada como uma subexpressão. Comparando a Fig. 2.20 com a árvore gramatical correspondente da Fig. 2.2, notamos que as árvores sintáticas assemelham-se muito à árvore gramatical com os nós internos de fazer dos operadores em lugar de seus filhos.

É desejável que um esquema de tradução esteja baseado numa gramática, pois árvores gramaticais separam preocupações das sintaxes: a estrutura hierárquica de subexpressões pela gramática da Fig. 2.1 é crucial para o agrupamento de operadores e operandos. Infelizmente, a gramática da Fig. 2.19 é recursiva à esquerda e, conseqüentemente, inadequada à análise gramatical preditiva. Parece existir um conflito: por um lado, precisamos de uma gramática que facilite a análise, por outro, precisamos de uma gramática boa, diretamente correspondente à regra de tradução. A solução óbvia é eliminar a recursividade à esquerda. No entanto, isso pode ser feito ad hoc, como no exemplo seguinte adiante.

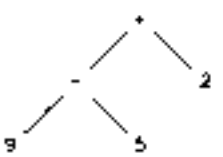


Fig. 2.20: Árvore sintática para `9-5+2`.

Exemplo 2.9. A gramática seguinte é modificada para a tradução de expressões na forma posfixa, ainda que a mesma gere exatamente a mesma linguagem que a gramática da Fig. 2.19. Esse pode ser usada para a análise gramatical descendente recursiva.

```
expr   → termo resto
resto  → + termo { imprima('+') } resto
       | - termo { imprima('-') } resto
       | ε
termo  → 0 { imprima('0') }
       | 1 { imprima('1') }
       ...
       | 9 { imprima('9') }
```

Isso garante a possível problema de que os operandos dos operadores gerados por `expr → expr + termo` e `expr → expr - termo` não são óbvios a partir das produções. Nenhuma das seguintes alternativas para formar a tradução de `resto` a partir daquela de `expr` é aceitável.

```
resto → + termo { imprima('+') } resto | - termo { imprima('-') } resto | ε   (2.12)
resto → resto + termo { imprima('+') } | resto - termo { imprima('-') } | ε   (2.13)
```

Mostramos somente a produção e ação semântica para o operador `+`. A tradução de `expr - termo` é semelhante. Se usarmos a ação em (2.12) para o sinal de menos, aparecerá antes de `expr` e `termo` permanecerá, incorretamente, `-` na tradução.

Por outro lado, se usarmos (2.13), a regra analisará para `-` os operadores serão gerados corretamente para a tradução. `+` e `-` de duas maneiras diferentes. Infelizmente, a tradução correta de `1 - 2 + 3` é `1 2 - 3 +`.

Adaptando o Esquema de Tradução

A técnica de eliminação da recursão à esquerda de uma gramática pode ser aplicada às produções contendo ações semânticas. Estudaremos a transformação na Seção 5.5 a fim de levarmos os atributos mencionados em conta. A técnica transforma as produções $A \to A\alpha \mid \beta$ em

$$A \to \beta R$$
$$R \to \alpha R \mid \varepsilon$$

Quando as ações semânticas estão intercaladas nas produções, carregam-las na transformação. Atribuindo $A = \text{expr}$, $\alpha = +\ \text{termo}\ \{\text{imprima}('+')\}$, $\beta = \text{termo}\ \{\text{imprima}('-')\}$ e $\gamma = \text{termo}$ e transformando a primeira produção do esquema de tradução (2.14). As produções de `expr` na Fig. 2.19 tornam-se as seguintes em (2.14). As produções para `termo` são as mesmas em ambos os esquemas. As produções para `resto` são as especificadas para a Fig. 2.19. Note que a gramática subjacente é diferente daquela do Exemplo 2.9 e que a tradução também é diferente daquela possível.

```
expr   → termo resto
resto  → + termo { imprima('+') } resto
       | - termo { imprima('-') } resto
       | ε
termo  → 0 { imprima('0') }                                   (2.14)
       | 1 { imprima('1') }
       ...
       | 9 { imprima('9') }
```

A Fig. 2.21 mostra como `9-5+2` é traduzida usando-se a gramática acima.

Procedimentos para os Não-Terminais

expr, termo e *resto*

Implementamos agora um tradutor em C usando o esquema de tradução dirigida pela sintaxe (2.14). A especificação do tradutor encontra-se na Fig. 2.22 sob as linhas `%{ ... %}` e `%% ... %%`. Essas linhas incluem ações semânticas associadas com as produções não-terminais correspondentes em (2.14).

A função `yylex()`, apresentada mais tarde, é a contraparte em C do analisador da Fig. 2.17 para reconhecer números e operadores.

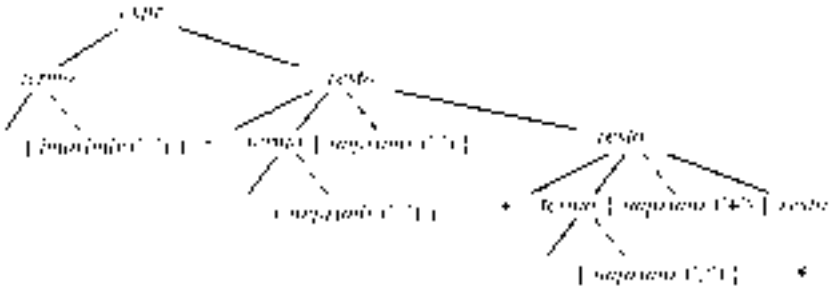


Fig. 2.21 Tradução de

pelo *backtracking* e avançar através da entrada. Como cada *token* é um caractere em nossa linguagem, [illegible] pode ser implementada comparando e lendo caracteres.

Para aqueles mais familiarizados com a programação na linguagem C, mencionamos as diferenças mais salientes entre C e as duas linguagens de teste, isto é, Algol-like e um Pascal-like, mencionadas anteriormente. Essas são aquelas ligeiramente de C. Um programa em C consiste em uma sequência de definições de funções e declarações, intercaladas uma após a outra, sem uma função chamada [illegible]. Definições de funções não podem ser aninhadas. Parâmetros recebidos pelas funções são sempre passados por valor, a menos que sejam explicitamente declarados como ponteiros. Assim, [illegible] são sempre passados por valor. As funções se comunicam transmitindo parâmetros por valor ou tendo acesso a dados globais a todas as funções. Por exemplo, as funções [illegible] examinam e modificam a variável global [illegible].

Use Pascal-like os seguintes símbolos para atribuição e testes de igualdade.

Operação	C	Pascal-like
atribuição	=	:=
teste de igualdade	==	=
teste de desigualdade	!=	<>

[illegible]

Fig. 2.22. Funções para [illegible]

As funções para os não-terminais testam os lados direitos de produções. Por exemplo, a produção para *expr* [illegible] é implementada pela chamada [illegible] na função [illegible].

Como um outro exemplo, a função [illegible] usa a primeira produção para *term* em (2.14), se o símbolo *lookahead* for um sinal de adição, a segunda, se for um sinal de subtração, e a produção *term* → *factor* por *default*. A próxima produção para *term* é implementada pelo primeiro enunciado [illegible] na Fig. 2.22. Se o símbolo *lookahead* for [illegible], o sinal de adição é reconhecido pela chamada [illegible]. Após a chamada [illegible], a rotina da biblioteca padrão de C [illegible] implementa a ação semântica, imprimindo o caractere [illegible]. Como a retenha produção para *term* possui o mesmo lado direito, o último enunciado [illegible] não faz coisa alguma.

As dez produções para *digit* geram os dez dígitos. Na Fig. 2.22, a ação [illegible] testa se o símbolo *lookahead* é um dígito. O dígito é impresso e reconhecido se o teste tiver sucesso, de outra forma uma chamada [illegible] indica o símbolo *lookahead* atual como uma impressão. Observe [illegible] antes do dígito, sob o escopo [illegible]. Antes de mostrar um programa completo, faremos uma transformação para melhorar a velocidade do código da Fig. 2.22.

Otimizando o Tradutor

Chamadas recursivas podem ser substituídas por iterações. Quando o último enunciado executado em um procedimento é uma chamada recursiva para o mesmo procedimento, a mesma é chamada de *epílogo recursivo*. Por exemplo, as chamadas [illegible] na quarta e última linha da função [illegible] são epílogo recursivo porque o controle flui para o final do corpo da função após a última delas.

Podemos acelerar um programa substituindo a epílogo-recursividade pela iteração. Para um procedimento sem parâmetros, uma chamada epílogo recursiva pode ser simplesmente substituída por um desvio para o início do procedimento. O código para [illegible] pode ser reescrito como:

[illegible]

No método em que o símbolo *lookahead* seja um sinal de adição ou de subtração, o procedimento [illegible] reconhece o sinal, chama [illegible] e, em seguida, [illegible] reconhece um dígito e repete o processo. Note que o enunciado [illegible] remove o sinal a cada vez em que é chamado, e, dessa forma, só permite sequências alternantes de sinais e dígitos. Se essa mudança

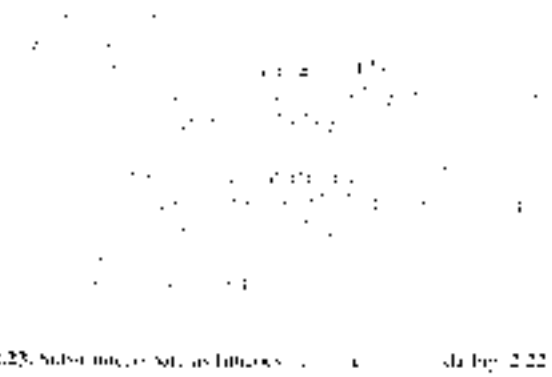


Fig. 2.23. Subrotinas ... da Fig. 2.22

... pois ... da Fig. 2.22, a única chamada ... para ... será proveniente de ... (linha 3). As duas funções podem ... sem ... serão agrupadas numa única, como mostrado na Fig. 2.23. Um ... encontrado, *getchar* pode ser ... respectivamente, ...

```
... void
```

... pois ... condição ... sempre verdadeira. Podemos sair de um ... executando um ... A versão ... da Fig. 2.23 permite que outros ... sejam ... sem ...

O Programa C Completo

O programa C completo para ... é mostrado na Fig. 2.24. A primeira linha ... um ... um arquivo de rotinas para ... O ... pela ... linha 2 da Fig. 2.24 a fim de antecipar ... que não ... são ... pela ... Como *lookahead* é ... de qualquer função, será global a qualquer uma que ... após a linha 2 da Fig. 2.24.

A função ... de entrada ... o símbolo *lookahead*. Em ... encontrados ... de erro em ...

A função ... a função da biblioteca ... a fim de emitir a mensagem ... e ... Em ... termina a execução através de uma ... para outra função de biblioteca padrão.

Fig. 2.24. Programa ... produz ... em ... sequência ...

2.6 ANÁLISE LÉXICA

Remoção de Espaços em Branco e Comentários

Constantes

Reconhecendo Identificadores e Palavras-Chave

(2.15)

(2.16)

Interface para o Analisador Léxico

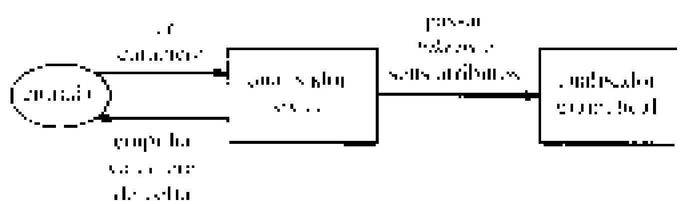


Fig. 2.25

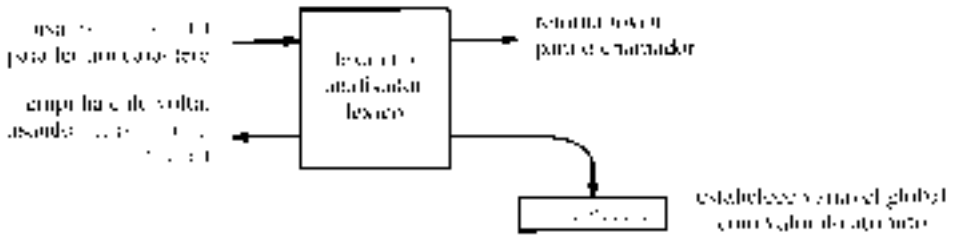


Fig. 2.26 Implementando as interações da Fig. 2.25

valor lexical será um ponteiro eliminado pelo gerador e armazenado as tabelas por demanda.

A implementação da leitura e classificação dos caracteres é usualmente feita através do estabelecimento de um *buffer* de entrada. Um bloco de caracteres é lido no *buffer* cada vez que um apontador ultrapassa a posição da última posição da entrada que foi analisada. A classificação dos caracteres é implementada através de uma tabela de apontadores. Os caracteres de entrada também precisam ser salvos para o relatório de erros, pois algum indicação pode ser devolvida sobre onde no texto de entrada ocorreu um erro. A linearização dos caracteres de entrada pode ser implementada independentemente da estrutura. A obtenção de todo um bloco de caracteres é usualmente mais eficiente do que a obtenção de um caractere de cada vez. As técnicas para linearização da entrada são discutidas na Seção 4.2.

Um Analisador Léxico

Construiremos a seguir um analisador léxico rudimentar para o tradutor de expressões da Seção 2.5. O propósito do analisador léxico é isolar o resto do tradutor de espaços em branco, números, palavras-chave e identificadores das expressões. Na próxima seção, estenderemos o analisador léxico de forma a também aceitar identificadores.

A Fig. 2.26 sugere como o analisador léxico escrito como uma função em C implementa as interações da Fig. 2.25. As variáveis [illegible] contêm cópias da localização da entrada. [illegible] empilha de volta os caracteres e o ponteiro para a entrada, chamando as rotinas [illegible] respectivamente. Como é declarada como um caractere, o par de enumerados

[illegible]

deriva do tipo de entrada indicado. A chamada para [illegible] armazena o próximo caractere de entrada a ser lido em [illegible]. A chamada para [illegible] empilha de volta o valor de [illegible] na entrada, fazendo com que ele seja o próximo caractere a ser lido.

Se a linguagem de implementação não permite que estruturas de dados sejam retornadas por funções, então os *tokens* e seus atributos precisam ser passados separadamente. A função [illegible] armazena em [illegible] uma especificação interna de um valor. O valor para um caractere pode ser qualquer codificação interna convencional daquele caractere. Um valor, como **num**, pode ser codificado por um valor inteiro maior que qualquer valor que codifique um caractere, digamos [illegible]. A fim de permitir que a codificação seja modificada facilmente, usaremos a constante simbólica [illegible] a fim de referenciarmos a codificação inteira de **num**. Em Pascal, a associação entre NUM e sua codificação pode ser feita por uma declaração constante; em C, [illegible] pode ser feita digitando-se [illegible] ou usando-se um comando *define*:

[illegible]

A função [illegible] retorna 1 quando uma sequência de dígitos é encontrada na entrada. A variável global [illegible] é estabelecida com o valor da sequência de dígitos. Dessa forma, serão 7 for seguido pelo caractere 8, [illegible] será 78. Se o caractere 9 for seguido pelo caractere A, [illegible] será 9. Se o primeiro caractere não for um dígito, [illegible] retorna 0.

Para uma linguagem de expressões regulares, requer uma modificação na gramática da Fig. 2.19. Sempre que um dígito não for precedido pelo símbolo de operação, [illegible] as seguintes produções e ações semânticas:

[illegible]

O código C para [illegible] é mostrado na Fig. 2.27. A variável [illegible] é usada para armazenar o valor de [illegible] entre chamadas. Cada vez que [illegible] é chamada, [illegible] é multiplicado por 10 e o valor de [illegible] é adicionado a ele. Assim, se [illegible] for 1 e [illegible] for 2, [illegible] será 12. Se [illegible] for 12 e [illegible] for 3, [illegible] será 123. Observe que [illegible] é usada para imprimir esse valor e também pela função de biblioteca [illegible]. O primeiro argumento de [illegible] é uma cadeia de caracteres especificando o formato a ser usado para [illegible] os argumentos remanescentes. Quando [illegible] aparece na cadeia de representação decimal do próximo argumento a ser impresso. Dessa forma, o comando [illegible] da Fig. 2.27 imprime um espaço seguido pela representação decimal de [illegible] seguida por outro espaço.

A implementação da função [illegible] é mostrada na Fig. 2.28. A cada vez que o ponteiro de entrada [illegible] avança nas linhas 8-28 é executado, um caractere é lido em [illegible] na linha 9. Se o caractere for um espaço em branco, tabulação ou nova linha, [illegible] avança sobre ele. Se o caractere for o fim de arquivo, [illegible] é retornado. Se o caractere for um dígito, [illegible] é chamada para [illegible] o valor numérico do dígito. Se o caractere não for nenhum dos anteriores, [illegible] é retornado. Observe que [illegible] é mantido dessa forma o contexto dos números de linha na entrada para, de novo, emitir mensagens de erro. O gerenciamento de uma mensagem de erro como o número da linha atual na localização dos erros.

O código para a leitura de uma sequência de dígitos está nas linhas 14-21. O predicado [illegible] é verdadeiro se [illegible] for um dígito decimal. As instruções nas linhas 14-17 afirmam se o caractere recém-chegado for um dígito, seu valor numérico será calculado pela expressão [illegible] e o dígito será lido. Como muitos conjuntos de caracteres, pode ser necessário que a conversão seja feita diferentemente. Na Seção 2.9 incorporaremos esse analisador léxico no nosso tradutor de expressões.

[illegible]

Fig. 2.27 Código em C para [illegible] e [illegible] e [illegible] números inteiros.

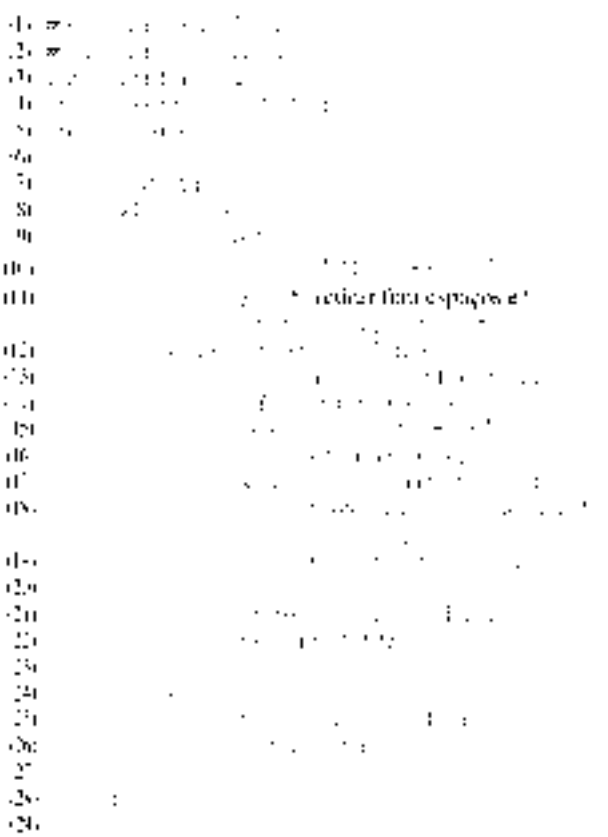


Fig. 2.28. Código para o analisador léxico que elimina espaços em branco e calcula números.

2.7 INCORPORANDO UMA TABELA DE SÍMBOLOS

Uma estrutura de dados, chamada de tabela de símbolos, é geralmente usada para armazenar informações sobre as várias construções da linguagem-fonte. A informação é coletada pelas fases de análise do compilador e usada pelas fases de síntese de forma a gerar o código-objeto. Por exemplo, durante a análise léxica, a cadeia de caracteres ou lexema que forma um identificador é guardada numa entrada da tabela de símbolos. As fases posteriores do compilador poderiam acrescentar informações à entrada, tais como o tipo do identificador, seu escopo e, provavelmente, variável ou rotina e sua posição de armazenamento. A fase de geração de código usaria as informações para gerar código apropriado para armazenar dados e dar acesso à memória. Na seção 2.8 discutiremos a implementação e uso das tabelas de símbolos em detalhes. Nesta seção, destacamos como o analisador léxico e o gerador de analisadores podem interagir com a tabela de símbolos.

A Interface com a Tabela de Símbolos

As rotinas da tabela de símbolos estão relacionadas principalmente com o armazenamento e recuperação dos lexemas. Quando um lexema é armazenado, também é chamado de *atributo* associado ao mesmo. As seguintes operações serão realizadas na tabela de símbolos:

- `inserir(s, t)`: Retorna o índice da nova entrada para a cadeia *s*, de token *t*.
- `pesquisar(s)`: Retorna o índice de uma entrada para a cadeia *s*, ou 0 se *s* não se encontrar na tabela.

Quando `pesquisar(s)` executa uma operação de busca para determinar se existe uma entrada para uma cadeia *s*, a tabela de símbolos é necessária, pois não é possível acessar uma entrada até que se saiba onde ela está. Discutiremos uma implementação na qual o analisador léxico envia cada cadeia, antes, à função de busca da tabela de símbolos.

O Tratamento das Palavras-Chave Reservadas

As rotinas da tabela de símbolos podem tratar qualquer coleção de palavras-chave reservadas. Como exemplo, considere-se uma tabela de símbolos que contém os lexemas `div` e `mod` com os tokens `div` e `mod`, respectivamente. Poderíamos inicializar a tabela de símbolos usando as chamadas

```
inserir("div", div)
inserir("mod", mod)
```

Qualquer chamada subsequente `pesquisar("div")` retornaria `div`, de tal forma que `div` não pode ser mais usado como um identificador.

Qualquer coleção de palavras-chave reservadas pode ser tratada desta forma, através da criação apropriada da tabela de símbolos.

Uma Implementação da Tabela de Símbolos

A estrutura de dados para uma implementação particular da tabela de símbolos é delineada na Fig. 2.29. Não desejamos restringir um lexema a um pequeno número fixo de espaço para guardar os lexemas que formam os identificadores, uma vez que uma quantidade fixa de espaço pode não ser suficiente para identificadores muito longos e pode ser grande demais para um identificador curto, como `i`. Na Fig. 2.29, o array isolado `lexemas` guarda as cadeias de caracteres que formam os identificadores. Cada cadeia é terminada por um caractere de fim de cadeia, denotado por EOS (*end of string*), o qual não pode formar nos identificadores. Cada entrada no array da tabela de símbolos [illegible]

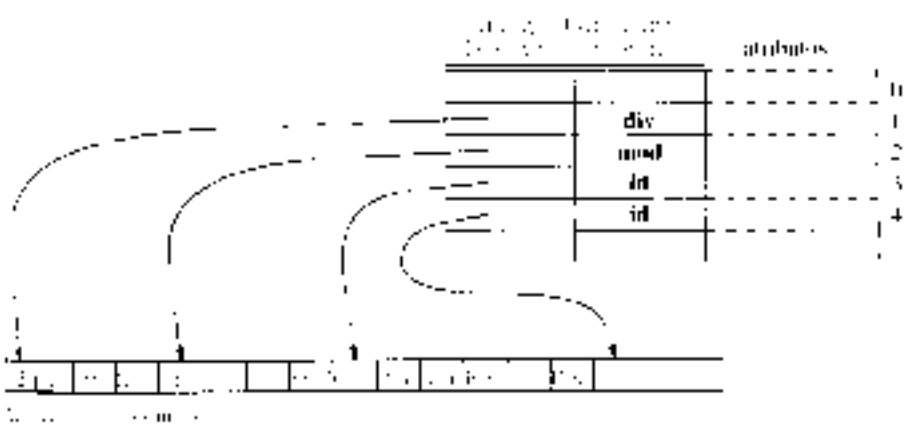


Fig. 2.29. A tabela de símbolos e o array para armazenar cadeias.

[illegible] consistindo em dois campos [illegible] que aponta para o nível de aninhamento [illegible]. Campos adicionais são usados para armazenar valores dos atributos, apesar de nenhuma dessas [illegible].

No fig. 2.29 a [illegible] é usada [illegible] para [illegible] retornar ao fim de indicação que não existe entrada para uma variável [illegible]. A pesquisa segue [illegible] para os valores [illegible].

A função [illegible] para [illegible] mostrados na fig. 2.30, uma implementação em Clipper do Seção 2.9. Os [illegible] manipulação dos [illegible] são [illegible] pelo analisador léxico da mesma [illegible] fig. 2.28 na [illegible].

Quando [illegible] léxico presente [illegible] letra, começa por armazenar letras e dígitos subsequentes no *buffer* [illegible]. A variável [illegible] é [illegible] na tabela de símbolos, através [illegible]. Como a tabela de símbolos é inicializada com variáveis para as palavras-chave [illegible], como mostrado na fig. 2.29, a operação básica [illegible] contém [illegible]. Se [illegible] entrada, para [illegible] se [illegible] é [illegible] uma entrada [illegible]. Uma variável [illegible] é criada usando-se [illegible].

Após a inserção [illegible] é [illegible] da entrada da tabela de símbolos para a variável [illegible]. Esse índice é armazenado [illegible] através [illegible] atributo [illegible]. Se [illegible] é [illegible] para [illegible].

A seção [illegible] retorna a [illegible] como [illegible] e [illegible] atributos [illegible] estabelecido como [illegible].

```
função Lexer: inteiro;
var lexema: array [0..100] de caracteres;
    c: caracteres;
início
  repetir lendo
    ler um caractere em c;
    se c é um espaço em branco, uma marca de tabulação ou um
      ponto de [illegible]
    senão se c for um caractere de avanço de linha então
      linha := linha + 1
    senão se c for um dígito então início
      fazer valor igual [illegible] dígitos [illegible]
      retornar
    fim
    senão se c for uma letra então início
      reunir [illegible] e dígitos subsequentes em lexema;
      p := [illegible](lexema);
      se p = 0 então
        p := [illegible];
      valor := p;
      retornar o componente [illegible]
    fim
    senão início { o token é um único caractere }
      fazer valor igual [illegible] atributo [illegible]
      retornar a codificação inteira [illegible]
    fim
  fim
fim
```

Fig. 2.30. Pseudocódigo para um analisador léxico.

2.8 MÁQUINAS DE PILHA ABSTRATAS

[illegible] de um compilador consiste [illegible] representação intermediária do programa fonte a partir da qual [illegible] de relacionamento da geração [illegible]. Uma forma [illegible] de representação intermediária [illegible] para [illegible] *máquina de pilha abstrata*. Como [illegible] mostrado no Cap. 1, [illegible] um compilador numa [illegible] de vários [illegible] de uma delas [illegible] para que [illegible].

Nesta seção, apresentamos [illegible] máquina de pilha abstrata e mostramos como o código para a mesma pode ser gerado. A máquina possui [illegible] de dados e de instruções separadas e todas [illegible] operações aritméticas [illegible] sobre valores [illegible] pilha. As instruções são [illegible] em três classes: aritmética [illegible], manipulação de pilha e [illegible] de controle. A Fig. 2.31 [illegible] máquina. O [illegible] do programa [illegible] a instrução que estamos prestes a executar. Os significados das instruções [illegible] são discutidos [illegible].

Instruções Aritméticas

A máquina de pilha abstrata precisa implementar cada operador da linguagem intermediária. Uma operação binária, [illegible] adição ou subtração, é [illegible] diretamente pela máquina [illegible]. Para operações mais complexas, envolvendo, [illegible] necessitar [illegible] sequências de instruções da máquina abstrata. Simplificamos a descrição [illegible] máquina assumindo que existe uma instrução para cada operador [illegible].

O código da máquina abstrata para uma expressão armazena o valor da [illegible] de uma representação pós-fixa para [illegible] na pilha. As [illegible] da [illegible] de processamento da representação pós-fixa, da esquerda para a direita, empilhando cada operando [illegible] que seja encontrado. Quando um operador é encontrado, seus argumentos estão [illegible] no topo da pilha e [illegible] pelos [illegible] operandos. As [illegible] após [illegible] operadores [illegible] pilha [illegible] pilha. Por exemplo, [illegible] da expressão pós-fixa [illegible] as seguintes ações são realizadas:

1. Empilhar 1
2. Empilhar 3
3. Substituir os dois elementos superiores da pilha desempilhando-os e empilhando o resultado 4
4. Empilhar 5
5. Multiplicar os dois elementos superiores da pilha desempilhando-os e empilhando o resultado 20

O valor no topo da pilha [illegible] exemplo, 20 é o valor de toda a expressão.

Na linguagem intermediária todos os valores serão inteiros com [illegible] operadores [illegible] meios diferentes de [illegible].

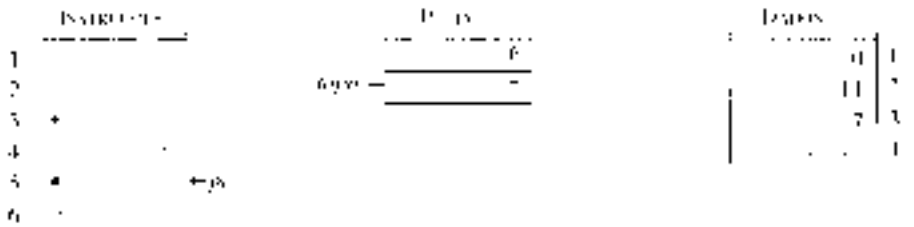


Fig. 2.31. [illegible] máquina de pilha [illegible] das [illegible] instruções.

Os operadores podem ser [illegible] que representam atributos, cujos valores já foram calculados.[1]

Valores-L e Valores-R

Lembramos da discussão entre os significados de identificadores à esquerda e à direita de uma atribuição (ver a Seção 2.7.3) as atribuições

```
[illegible]
```

atribuem valores específicos a um identificador, enquanto que o identificador à esquerda do operador de atribuição deve ser interpretado simbolicamente, como um operador sobre o endereço de uma localização:

```
[illegible]
```

Assim, o operador `id` à esquerda do operador de atribuição `=` deve ser interpretado como um endereço, enquanto que o `id` à direita deve ser interpretado como um valor. Os termos *valor-l* e *valor-r* são abreviações de "valor à esquerda" e "valor à direita", respectivamente, e podem ser interpretados como "endereço" e "valor", respectivamente. Nem todos os valores que aparecem à esquerda de uma atribuição são endereços. Por exemplo, em `i = 3`, o valor de `i` é o endereço onde o valor de 3 será armazenado. Similarmente, `a[2] = 3` significa que o valor de `a[2]` é o endereço onde o valor de 3 será armazenado. Um valor-l é um endereço; um valor-r é um valor, que pode ser um endereço.

Em uma máquina de pilha, o valor-l de um identificador é o endereço do identificador. O valor-r de um identificador é o valor contido no endereço do identificador. Quando um identificador aparece à esquerda de uma atribuição, ele é avaliado para um endereço. Quando ele aparece em uma expressão, ele é avaliado para o valor contido em seu endereço. Por exemplo, a atribuição

```
[illegible]
```

é traduzida no código da Fig. 2.37.

Esses acessos também podem ser expressos formalmente como segue. Declara-se `top` como um ponteiro para a pilha, que contém o endereço do topo da pilha. O atributo `lexeme` de `id` fornece a representação do identificador sob a forma de cadeia de caracteres:

```
[illegible]
```

Fluxo de Controle

A máquina de pilha executa instruções em sequência, a menos que seja orientada a agir de outra maneira através de um comando de desvio condicional ou incondicional. Vamos acrescentar três instruções para especificar o destino dos desvios:

1. O operando da instrução fornece a localização alvo.
2. O operando da instrução fornece a distância relativa, positiva ou negativa, a ser saltada.
3. O destino é especificado simbolicamente como um rótulo. Uma máquina de pilha precisa manter uma tabela que mapeia rótulos em endereços de máquina.

Cada uma dessas três opções oferece a possibilidade de especificar o destino de um desvio. Sofisticados endereços simbólicos não são necessários quando estamos apenas gerando código para a máquina alvo. Fizemos certos melhoramentos no código que resultou na seção com a tradução de atribuições.

As instruções para o fluxo de controle em uma máquina de pilha são:

	desvia o valor dos dois operandos e não passa nada adiante
	[illegible]
	[illegible]
	desempilha o valor do topo da pilha; desvia se o valor for zero
	[illegible]

Tradução de Enunciados

O algoritmo na Fig. 2.35 definiu a tradução para a maioria dos enunciados condicionais e de repetição. A seguinte discussão se restringe à tradução de rótulos.

Vamos descrever o código para o enunciado `if` na Fig. 2.37. Suponha que haver uma instrução [illegible] em todo programa fonte; de outra forma haveria confusão sobre para onde vai o

[illegible]

Fig. 2.32. Tradução de [illegible]

[1] Na verdade, o operador `=` é tratado como um operador especial, com propriedades especiais, como discutido em [illegible].

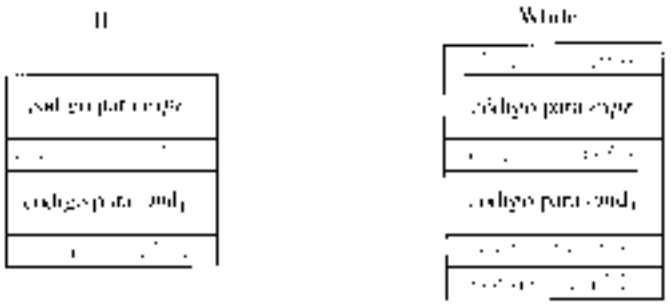


Fig. 2.33. Geração de código para comandos com desvios e repetições

nos devolva linha a partir de um enunciado [illegible]. Precisamos também de algum mecanismo para emitir sucessivamente rótulos [illegible] no gabarito do código por um procedimento cada vez que uma instrução for traduzida.

Suponhamos que novo_rótulo seja um procedimento que retorna um rótulo novo cada vez que seja chamado. Na seguinte tradução esquemática, o rótulo retornado por uma chamada de novo_rótulo é registrado usando uma variável local rótulo:

```
cond → if expr then cond1   { rótulo = novo_rótulo();
                              cond.c = expr.c
                                       || 'if' || expr.t || 'goto' || rótulo
                                       || cond1.c
                                       || rótulo || ':' }
```

Emitindo uma Tradução

Os tradutores de expressões da Seção 2.8 usavam enunciados impressos para gerar a tradução de uma expressão. Todos estes enunciados impressos são similares podem ser usados para emitir a tradução de comandos. Em vez de imprimir os enunciados, geramos um procedimento emite que faça de ocultar os detalhes do impressão. Por exemplo, emite pode encarregar-se pessoalmente de cada instrução da máquina abstrata precisar estar numa linha separada. Usando o procedimento emite podemos escrever o esquema de tradução da Fig. 2.19

```
cond → if
          expr   { rótulo1 = novo_rótulo(); rótulo2 = novo_rótulo();
                   emite('if', expr.t, 'goto', rótulo1);
                   emite('goto', rótulo2);
                   emite(rótulo1, ':') }
          then
          cond1  { emite(rótulo2, ':') }
```

Quando as ações aparecem dentro de uma produção, devemos considerar os elementos em cada direita como numa representação pré-ordem da árvore de análise. Para a produção acima, considera-se as ações e como se segue: as ações durante a análise gramatical de cond1 são executadas antes da última instrução [illegible]. Assim, as ações durante a análise gramatical de cond1 são executadas finalmente, a instrução [illegible] é emitida. Assumindo que as ações durante a análise gramatical de expr e de cond1 emitam o código para esses não terminais, as produções acima implementam a palavra de código da Fig. 2.33.

O pseudocódigo para a tradução de comandos condicionais e de atribuição é mostrado na Fig. 2.34. Como o atributo cond.next [illegible] por enquanto cond.next será o valor não alterado pelas chamadas aos procedimentos [illegible]. A geração de rótulos requer algum cuidado. Suponhamos que os rótulos na tradução são da forma [illegible]. O pseudocódigo mantém um contador nrótulos que conta o número de rótulos já gerados. Assim, cada declaração de variável local rótulo é [illegible]

```
procedimento cond
var rótulo, nrótulos: inteiros; { para os rótulos }
início
    se lookahead = id então início
        [illegible]
        [illegible]
    fim
    senão se lookahead = [illegible] then início
        [illegible]
        expr;
        rótulo = novo_rótulo;
        emite('if', 'goto', rótulo);
        [illegible]
        cond;
        [illegible]
    fim
        { o código para os enunciados restantes vem logo aqui }
    senão erro;
fim
```

Fig. 2.34. Pseudocódigo para a tradução de comandos

reter na variável que vem a ser o valor de rótulo e assim precisa ser escrita de forma a imprimir por si só dado um inteiro.

O gabarito de código para os enunciados repetitivos de linhas na Fig. 2.33 pode ser considerado um código de uma variável simples. A tradução de uma sequência de enunciados é simplesmente a concatenação dos enunciados e a sequência é deixada para o leitor.

A tradução das construções de uma expressão é uma tarefa similar àquela dos enunciados repetitivos e fácil. Ilustramos a construção do fluxo de controle nas expressões.

Exemplo 2.10. O analisador sintático da Seção 2.7 com ações traduz a condição da forma

```
if espaço or = tabulação then
```

Se c for um espaço, então claramente não é necessário testar se c é um caractere de tabulação porque a primeira igualdade implica que a condição seja verdadeira. A expressão

```
c = espaço or c = tabulação
```

pode ser implementada como

```
if c = espaço then true else c = tabulação
```

O valor pode verificar que o código seguinte implementa a operação or:

```
código para expr1          { coloca valor de expr1 no topo }
                           { da pilha }
copy                       { duplica valor no topo da pilha }
if_true L                  { salta se o valor for verdadeiro }
pop                        { retira o valor de expr1 }
código para expr2          { coloca valor de expr2 no topo }
                           { da pilha }
L:
```

Relembramos que as instruções [illegible] removem o valor no topo da pilha e, em seguida, implementam a geração de código dos comandos condicionais e de repetição. Pela criação do valor de expr1, assegura-se que o valor no topo da pilha seja verdadeiro se a instrução [illegible] levar a um salto.

2.9 JUNTANDO AS TÉCNICAS

Neste capítulo apresentamos um número de técnicas dirigidas pela sintaxe para construir uma tradução. Para sintetizar essas técnicas, colocamos, nesta seção, um programa C que funciona

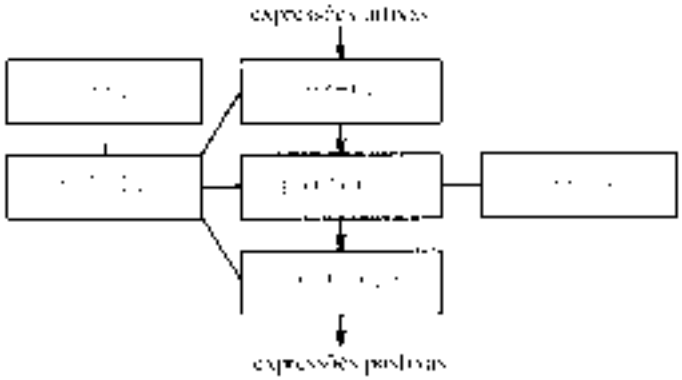


Fig. 2.36. Módulos para um tradutor Infixa para Pós-fixa.

Descrição do Tradutor

O tradutor a ser projetado usará o esquema de tradução dirigida pela sintaxe da Fig. 2.35. O token id representa uma sequência não vazia de caracteres alfanuméricos começando por uma letra. **num** representa uma sequência de dígitos e **eof** um caractere de fim de arquivo. Os tokens são separados por sequências de espaços, tabulações e avanços de linha ("espaços em branco"). O atributo lexval do token id fornece a cadeia de caracteres que forma o nome do identificador. O atributo lexval do token **num** fornece o valor representado por num.

O tradutor está organizado em seis módulos, cada um armazenado num arquivo separado. A execução começa no módulo principal, que consiste em uma chamada para para a inicialização seguida por uma chamada para para a tradução. Os seis módulos restantes são mostrados na Fig. 2.36. Existe também um arquivo de cabeçalho global que contém definições comuns a muitos dos módulos. O número enumerado em cada módulo

indica os arquivos de cabeçalho incluídos como parte do módulo. As seções restantes discutem o código para o tradutor descrevendo brevemente cada um deles e como eles foram construídos.

O Módulo de Análise Léxica

O analisador léxico é uma rotina chamada pelo parser a fim de encontrar tokens. Implementada a partir do pseudo-código da Fig. 2.30, a rotina lê a entrada um caractere de cada vez, retornando um token para cada lexema encontrado. O valor do atributo associado a um token é atribuído à variável global

Os seguintes tokens são esperados pelo parser:

Aqui representa um identificador, um número, e assim por diante. Os espaços em branco são simplesmente removidos pelo analisador léxico. A tabela na Fig. 2.37 mostra o token e o valor de atributo produzidos pelo analisador léxico para cada lexema de linguagem fonte.

```
num      →  [0-9]+
id       →  [a-z][a-z0-9]*
white    →  ( ' ' | '\t' | '\n' )+
%%
{white}  {/* ignora espaços em branco */}
{num}    {yylval = atoi(yytext); return num;}
{id}     {yylval = strdup(yytext); return id;}
"+"      {return '+';}
"-"      {return '-';}
"*"      {return '*';}
"/"      {return '/';}
"("      {return '(';}
")"      {return ')';}
<<EOF>>  {return eof;}
.        {printf("caractere inválido\n");}
```

Fig. 2.35 Especificação para um analisador léxico.

O analisador lexical usa a função da tabela de símbolos a fim de determinar se o lexema de um identificador já foi anteriormente examinado e a rotina para armazenar um novo lexema na tabela. Igualmente, ele retorna uma variável global a cada vez que encontra um caractere de avanço de linha.

O Módulo Parser

O parser é construído usando as técnicas da Seção 2.5. Primeiro eliminamos a recursão à esquerda do esquema de tradução da Fig. 2.35 de tal forma que a gramática subjacente possa ser analisada por um parser descendente recursivo. O esquema transformado é mostrado à Fig. 2.38.

Construímos, então, funções para os não-terminais

como mostra a Fig. 2.21. A função parse implementa o símbolo de partida da gramática, chama sempre que necessita de um novo token, e gera a saída usando a função para gerar a saída e a função para relatar um erro de sintaxe.

O Módulo Emissor

O módulo emissor consiste em uma única função que imprime o valor de seu argumento seguido por um espaço em branco.

Módulos da Tabela de Símbolos e

O módulo da tabela de símbolos implementa a estrutura de dados mostrada na Fig. 2.29 da Seção 2.7. As entradas na tabela são strings, e um índice inteiro denota uma string armazenada. A operação

LEXEMA	TOKEN	VALOR DO ATRIBUTO
espaço em branco		
sequência de dígitos		valor numérico da sequência
sequência de uma letra seguida por letras e dígitos		índice para tabela_sim
qualquer outro caractere		

Fig. 2.37. Descrições de tokens.

```
início → lista eof
lista → expr ; lista
      | ε
expr → termo maistermos
maistermos → + termo { emitir('+') } maistermos
      | - termo { emitir('-') } maistermos
      | ε
termo → fator maisfatores
maisfatores → * fator { emitir('*') } maisfatores
      | / fator { emitir('/') } maisfatores
      | div fator { emitir('DIV') } maisfatores
      | mod fator { emitir('MOD') } maisfatores
      | ε
fator → ( expr )
      | id { emitir(id, lexval) }
      | num { emitir(num, lexval) }
```

Fig. 2.38. Esquema de tradução dirigida pela sintaxe após a eliminação da recursão à esquerda.

[illegible] retorna o índice da entrada da tabela de símbolos para o lexema que [illegible]. A função [illegible] retorna o índice da entrada [illegible] para o lexema [illegible] se o mesmo ainda não estiver lá.

O módulo [illegible] é usado para pré-carregar a tabela de símbolos com palavras-chave. [illegible] todas as palavras-chave são armazenadas [illegible] e quais [illegible] no tipo que [illegible]. A função [illegible] percorre sequencialmente [illegible] da função [illegible] colocar as palavras-chave na tabela de símbolos. [illegible] em uma forma conveniente.

O Módulo de Erros [illegible]

O módulo de erros gerencia o relatório de erros [illegible] primitivo. Ao encontrar um erro de sintaxe, o compilador imprime uma mensagem [illegible] que um erro ocorreu na linha corrente de entrada e [illegible] para. Uma técnica melhor de recuperação de erros poderia saltar até o próximo ponto-e-vírgula e continuar [illegible] gramatical [illegible] encorajamos a realizar essa mudança no tradutor. Técnicas mais sofisticadas de recuperação de erros são apresentadas no Capítulo 4.

Criando o Compilador

O código para os [illegible] aparece em sete arquivos [illegible] [illegible] O [illegible] contém o módulo principal do programa C que chama [illegible] e [illegible] [illegible]

Sob o sistema operacional UNIX, o compilador pode ser criado através da execução do comando

[illegible]

ou compilando-se separadamente os arquivos, usando

[illegible]

e ligando-se os arquivos [illegible] resultantes

[illegible]

O comando [illegible] cria um arquivo [illegible] que contém o tradutor. O tradutor pode então ser exercitado digitando-se [illegible] seguida pelas expressões a serem traduzidas; por exemplo,

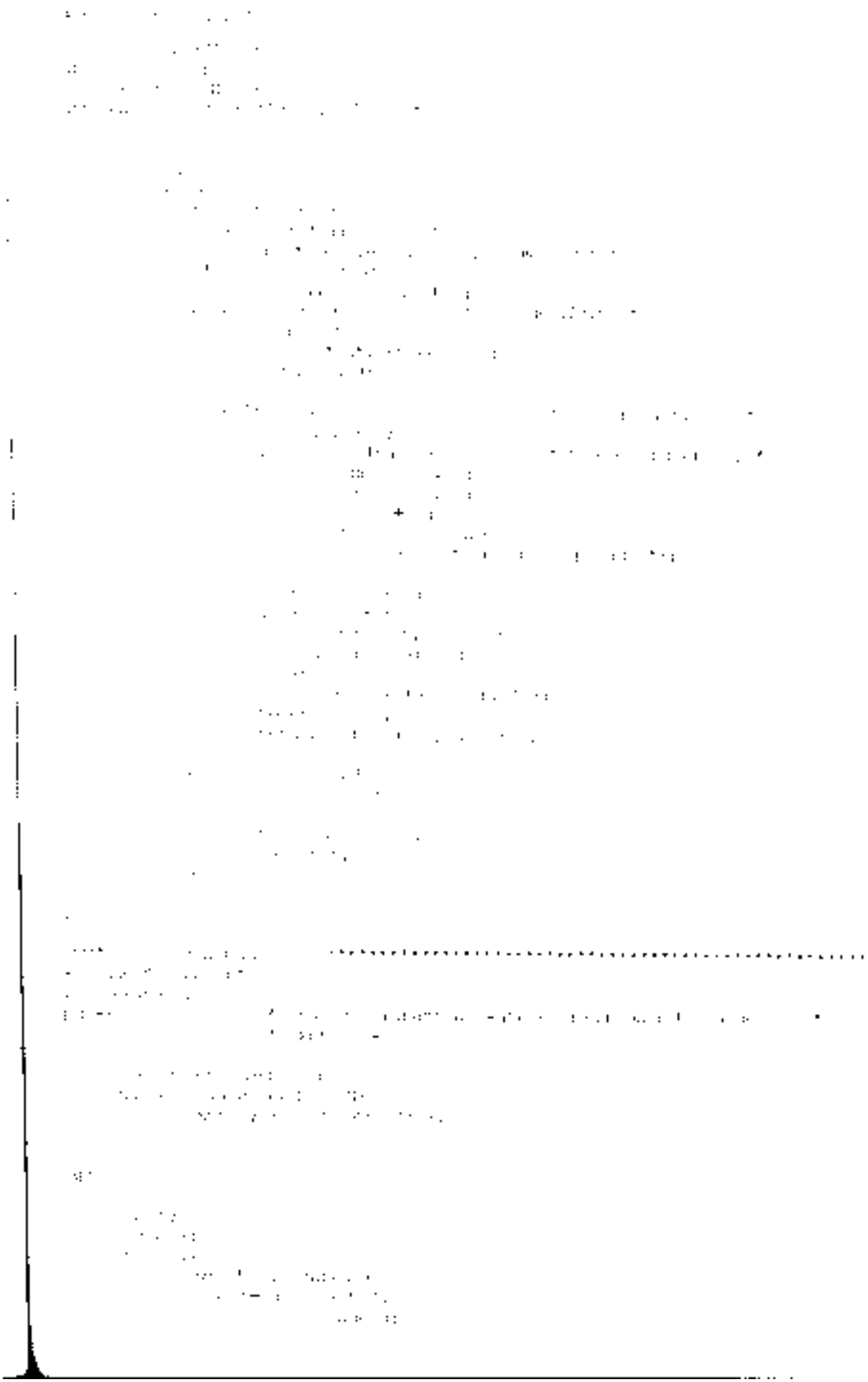
[illegible]

ou qualquer outra expressão de sua escolha. Tente.

A Listagem

Apresenta-se a listagem do programa C que implementa o tradutor. [illegible] arquivo global de cabeçalho [illegible] é mostrado, seguido pelos sete arquivos fonte. Por questões de clareza, o programa foi escrito num estilo elementar de C.

[illegible]



EXERCÍCIOS

2.1 Considere a gramática livre de contexto

$$S \rightarrow SS \mid (S) \mid \varepsilon$$

a) Mostre que a cadeia $(()())$ pode ser gerada por esta gramática.
b) Construa a árvore gramatical para essa cadeia.
c) Que tipo de linguagem é gerada por essa gramática? Justifique a sua resposta.

2.2 Que tipos de linguagens são geradas pelas seguintes gramáticas? Em cada caso justifique a sua resposta.
a) $S \rightarrow aS \mid a$
b) $S \rightarrow aS \mid aSbS \mid \varepsilon$
c) $S \rightarrow aSbS \mid bSaS \mid \varepsilon$
d) $S \rightarrow aSbS \mid bSaS \mid \varepsilon$
e) $S \rightarrow aSbS \mid bSaS \mid \varepsilon$

2.3 Quais das gramáticas no Exercício 2.2 são ambíguas?

2.4 Construa gramáticas livres de contexto não ambíguas para cada uma das seguintes linguagens. Em cada caso mostre que sua gramática está correta.
a) Expressões aritméticas na notação posfixa.
b) Listas não vazias à esquerda de identificadores separados por vírgulas.
c) Listas não vazias à direita de identificadores separados por vírgulas.
d) Expressões aritméticas de inteiros e identificadores com os quatro operadores binários $+, -, *, /$.
e) Incluindo os sinais e os menos unários aos operadores aritméticos do item (d).

***2.5** a) Mostre que todas as cadeias binárias geradas pela seguinte gramática possuem valores divisíveis por 3. *Dica:* Use indução no número de nós da árvore gramatical.

$$num \rightarrow 11 \mid num\ 00 \mid num\ 11 \mid 00 \mid num\ num$$

b) A gramática gera todas as cadeias binárias com valores divisíveis por 3?

2.6 Construa uma gramática livre de contexto para os números romanos.

2.7 Construa um esquema de tradução dirigida pela sintaxe que traduza expressões aritméticas da notação infixa para a prefixa, na qual um operador fica antes de seus operandos; por exemplo, $-xy$ é a notação prefixa para $x - y$. Forneça as árvores gramaticais anotadas para as entradas $(9 - 5) + 2$ e $9 - (5 + 2)$.

2.8 Construa um esquema de tradução dirigida pela sintaxe que traduza expressões aritméticas da notação posfixa para a infixa. Forneça as árvores gramaticais anotadas para as entradas $95 - 2 +$ e $952 + -$.

2.9 Construa um esquema de tradução dirigida pela sintaxe que traduza inteiros em números romanos.

2.10 Construa um esquema de tradução dirigida pela sintaxe que traduza números romanos em inteiros.

2.11 Construa *recursivos descendentes* (ou preditivos) para as gramáticas do Exercício 2.2 (a), (b) e (c).

2.12 Construa um tradutor dirigido pela sintaxe que verifique se os parênteses numa cadeia de entrada estão balanceados apropriadamente.

2.13 As seguintes regras definem a tradução de uma palavra inglesa para o *latim de porco*.
a) Se a palavra começa por uma cadeia não vazia de consoantes, mova a cadeia inicial de consoantes para o final da palavra e adicione *ay*. Assim, *pig* se torna *igpay*, e *trash* se torna *ashtray*.
b) Se a palavra começa por uma vogal, adicione o sufixo *yay*. Assim, *owl* se torna *owlyay*.
c) Separando um *y* de uma consoante.
d) As letras de uma palavra latina vogal se não for seguida por uma vogal.
e) Palavras com uma única letra não são alteradas.

Construa um esquema de tradução dirigida pela sintaxe para o *latim de porco*.

2.14 Na linguagem de programação C, o comando for possui a forma:

$$for\ (expr_1;\ expr_2;\ expr_3)\ stmt$$

A primeira expressão é executada antes do laço e geralmente usada para inicializar o índice do laço. A segunda expressão é um teste feito antes de cada iteração do laço, e o laço é abandonado se a expressão se torna 0. O laço em si consiste do comando $stmt$. A terceira expressão é executada ao fim de cada iteração e tipicamente usada para incrementar o índice de laço. O significado do comando **for** é similar a

$$expr_1;\ while\ (expr_2)\ \{\ stmt\ expr_3;\ \}$$

Construa um esquema de tradução dirigida pela sintaxe para traduzir *estruturas de laços for* em código de máquina de pilha.

***2.15** Considere o seguinte comando **for**

for $i = 1$ **step** 10 $\ldots$ **until** $10 * i + 1$ **do** $i = j - 1$

Três definições semânticas podem ser encontradas para esse enunciado. Um passo (step) significando que a função $10 \ldots$ no incremento $10 \ldots$ deve ser avaliada a cada iteração do laço, como em PL/I; *por exemplo*, se $j = 5$ antes do laço, ele irá executar nove vezes através do mesmo e sair. Um segundo significado, completamente diferente, seria o de que o valor de $10 * i + 1$ deve ser avaliado apenas uma vez, antes de o laço começar a ser executado. Um terceiro significado é dado por linguagens tais como Algol. *Quando* o incremento é negativo, o teste feito para terminação do laço é $<$ 10 ... em vez de $>$ 10 ... Para cada uma dessas definições semânticas, construa um esquema de tradução dirigida pela sintaxe para traduzir esses laços *for* em código de máquina de pilha.

2.16 Considere o seguinte fragmento de gramática para comandos *if-then* e *if-then-else*:

$$\begin{aligned} stmt \rightarrow\ & \textbf{if}\ expr\ \textbf{then}\ stmt \\ \mid\ & \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt \\ \mid\ & other \end{aligned}$$

sendo **other** usado no lugar de outros comandos da linguagem.
a) Mostre que essa gramática é ambígua.
b) Construa uma gramática equivalente não ambígua que associe cada **else** com o **then** mais próximo não associado.
c) Construa um esquema de tradução baseado nesta gramática que traduza os enunciados condicionais em código de máquina de pilha.

***2.17** Construa um esquema de tradução dirigida pela sintaxe que traduza expressões aritméticas na notação infixa em expressões aritméticas na notação infixa sem ter parênteses redundantes. Mostre a árvore gramatical anotada para a entrada $(a + b) * (c * (d + e))$.

EXERCÍCIOS DE PROGRAMAÇÃO

P2.1 Implemente os tradutores de inteiros para números romanos baseado no esquema de tradução dirigida pela sintaxe construído no Exercício 2.9.

P2.2 Modifique o tradutor da Seção 2.4 para que produza como saída código para a máquina abstrata de pilha da Seção 2.8.

P2.3 Modifique o módulo de geração de código da tradução da Seção 2.9 a fim de que saiba como gerar código para a previsão da expressão de controle no início do laço.

P2.4 Expanda o tradutor da Seção 2.9 a fim de incluir os comandos abaixo de código de máquina de pilha com variáveis gerados pela seguinte gramática:

$$\begin{aligned} cmd &\rightarrow \textbf{id} := exp \\ &\quad \textbf{if } exp \textbf{ then } cmd \\ &\quad \textbf{while } exp \textbf{ do } cmd \\ &\quad \textbf{begin } com\_cmd \textbf{ end} \\ exp &\rightarrow cmd \textbf{ first } exp \\ com\_cmd &\rightarrow com\_cmd\ ;\ cmd \mid cmd \end{aligned}$$

*P2.6 Construa um conjunto de expressões de teste para o compilador da Seção 2.9 de tal forma que cada produção seja usada pelo menos uma vez na derivação de alguma expressão de teste. Construa um programa de teste que possa ser usado como uma ferramenta geral de teste de compiladores. Use seu programa de teste para validar seu compilador em suas expressões de teste.

P2.7 Construa um conjunto de enunciados para seu compilador do Exercício P2.5, de tal forma que cada produção seja usada pelo menos uma vez a fim de gerar algum enunciado de teste. Use o programa de testes do Exercício P2.6 para validar seu compilador sobre essas expressões de testes.

NOTAS BIBLIOGRÁFICAS

Este capítulo introdutório foi uma miscelânea de assuntos que são tratados em muito mais detalhes pelo resto do livro. Remetemos a Lee [1967] e Opler [1967] para mais capítulos com conteúdo material posterior.

As gramáticas livres de contexto foram introduzidas por Chomsky [1956] como parte de um estudo das linguagens naturais. Seu uso na especificação da sintaxe das linguagens de programação foi sugerido independentemente. Enquanto trabalhava no projeto inicial do Algol 58, John Backus "aproximadamente adaptou as produções de [Emil Post] para aquele uso" (Wexelblat [1981], p. 162). A notação resultante foi uma variação das gramáticas livres de contexto. Posteriormente, Peter Naur introduziu uma notação sintática equivalente a fim de especificar as regras da gramática Algol 60 (ver Seção 4.6.1 e Naur et al. [1963]).

A proposição de que BNF, que começou como uma abreviatura de Backus Normal Form, seja lida como Backus–Naur Form, para reconhecer as contribuições de Naur como editor do relatório do Algol 60 (Naur [1963]), está contida numa carta de Knuth [1964].

As definições dirigidas pela sintaxe são formas de definições indutivas na qual a indução está na estrutura sintática. Como tal, têm sido usadas por bastante tempo informalmente na matemática. Suas aplicações às linguagens de programação chegam a ponto de uso de uma gramática para estruturar o relatório do Algol 60. Pouco depois, Ingerman [1961] construiu um compilador dirigido pela sintaxe.

A análise preditiva descendente recursiva tem sido usada desde o início dos anos 60. Hoare [1976] atribui o método a Conway [1963] e Hoare [1962, p. 128] descreve um compilador Algol organizado como um conjunto de procedimentos, cada um deles capaz de processar uma das unidades sintáticas do relatório Algol 60. Foster [1968] discute a eliminação da recursão e expande os procedimentos acrescentando ações semânticas que manipulam valores de atributos.

McCarthy [1963] advogou que a definição de uma linguagem deveria ser, sempre que possível, abstrata. No mesmo artigo, McCarthy [1963, p. 24] deixou o leitor com o desafio de escrever uma definição explícita da função fatorial e equivalente a um programa interativo.

Os benefícios de se dividir um compilador numa varredura e numa retaguarda foram explorados num relatório de comitê por Strong et al. [1958]. O relatório cunhou o nome UNCOL, proveniente de Linguagem Universal Orientada para Computadores — *Universal Computer Oriented Language*) para uma linguagem intermediária universal. O conceito permanece como um ideal.

Uma boa forma de se aprender as técnicas de implementação é ler os códigos de compiladores existentes. Infelizmente, poucos códigos são frequentemente publicados. Randell e Russell [1964] fornece uma contabilidade extensiva de um dos primeiros compiladores Algol 60. O código do compilador Pascal P (Nori et al. [1981]), detalhes de geração de código (Ammann [1977]), e o código para uma implementação de Pascal S, um subconjunto de Pascal projetado por Wirth [1981] para uso de estudantes, Kruseman [1985] fornece uma descrição minuciosamente clara e detalhada do tradutor TgA.

Kernighan e Pike [1984] descrevem em detalhes como construir um *yacc*-grama para simular uma calculadora de mesa e o sistema de um esquema de tradução dirigida pela sintaxe usando as ferramentas de construção de compiladores disponíveis no sistema operacional UNIX. A Equação (2.17), é proveniente de Tanzen [1963].

CAPÍTULO 3

A ANÁLISE LÉXICA

Este capítulo trata de técnicas para especificar e implementar analisadores léxicos. Uma forma simples de se construir um analisador léxico é escrever um diagrama que ilustre a estrutura léxica da linguagem fonte e então traduzir manualmente num programa que os localize. Analisadores léxicos eficientes podem ser construídos dessa forma.

As técnicas usadas para implementar os analisadores léxicos podem também ser aplicadas a outras áreas, tais como linguagens de interrogação a sistemas de recuperação de informações. Para cada aplicação o problema subjacente é a especificação e o projeto de programas que executam ações disparadas por padrões nas cadeias de caracteres. Como a programação orientada para o processamento de padrões é útil em geral, apresentamos uma linguagem de ações baseadas em padrões, chamada Lex, para a especificação de analisadores léxicos. Nessa linguagem, os padrões são especificados por expressões regulares, e um compilador Lex pode gerar automaticamente um reconhecedor para essas expressões regulares.

Várias outras linguagens usam expressões regulares para descrever padrões. Por exemplo, a linguagem de reconhecimento de padrões AWK usa expressões regulares para selecionar linhas de entrada para processamento. O shell do sistema UNIX permite que o usuário se refira a um conjunto de nomes de arquivos escrevendo uma expressão regular. (O comando UNIX `rm *.o`, por exemplo, remove todos os arquivos com nomes terminados em ".o".)

Uma ferramenta de software que automatiza a construção de analisadores léxicos permite que pessoas com diferentes formações usem o reconhecimento de padrões em suas próprias áreas de aplicação. Por exemplo, Lesk [1976] usou um gerador de analisadores léxicos para criar um programa que reconhecesse as imperfeições de placas de circuitos impressos. Os circuitos estão normalmente descritos em tabelas de segmentos de linha em ângulos diferentes. O "analisador léxico" procura pelos padrões que correspondem a essas imperfeições na tabela de segmentos de linha. Uma vantagem usar um gerador de analisadores léxicos é que o mesmo pode utilizar os algoritmos de reconhecimento de padrões mais conhecidos e desse forma criar analisadores léxicos eficientes para pessoas que não sejam especialistas em técnicas de reconhecimento.

3.1 O PAPEL DO ANALISADOR LÉXICO

O analisador léxico é a primeira fase de um compilador. Sua função principal é ler os caracteres de entrada e produzir uma seqüência de tokens que o parser utiliza para a análise sintática. Essa interação, sumarizada esquematicamente na Fig. 3.1, é comumente implementada fazendo-se com que o analisador léxico seja uma sub-rotina ou uma co-rotina do parser. Ao receber esse comando, o analisador léxico lê os caracteres de entrada até que possa identificar o próximo token.

Como o analisador léxico é a parte do compilador que lê o texto-fonte, ele também pode realizar algumas outras funções secundárias no nível da interface com o usuário. Uma delas é a de remover do programa-fonte os comentários e espaços em branco, isto é, os caracteres sob a forma de espaços, tabulações e caracteres de avanço de linha. Uma outra é a de correlacionar as mensagens de erro do compilador com o programa-fonte. Por exemplo, o analisador léxico pode controlar o número de caracteres novos de linha que foram lidos de tal forma que uma mensagem de erro possa ser relacionada a uma mensagem de erro. Em alguns compiladores, o analisador léxico tem a responsabilidade de fazer uma cópia do programa-fonte com as mensagens de erro associadas a ele mesmo. Se a linguagem-fonte suporta algumas funções sobre a forma de macros pré-processadas, as macros também podem ser implementadas na medida em que a análise léxica vai se desenvolvendo.

Algumas vezes, os analisadores léxicos são divididos em duas fases em cascata, a primeira chamada de "varredura" (scanning) e a segunda de "análise léxica". O scanner é responsável por realizar tarefas simples, enquanto o analisador léxico propriamente dito realiza as tarefas mais complexas. Por exemplo, um compilador Fortran poderia usar um scanner para eliminar espaços em branco.

Temas da Análise Léxica

Existem várias razões para se dividir a fase de análise de uma compilação em analisador léxico e analisador sintático (parser).

1. Um projeto mais simples quase sempre é a consideração mais importante. A separação das análises léxica e sintática frequentemente nos permite simplificar uma ou outra dessas fases. Por exemplo, um parser que incorpore as convenções para comentários e espaços em branco é significativamente mais complexo do que um que assume que os mesmos já tenham sido removidos pelo analisador léxico. Se estivermos projetando uma nova linguagem, separar as convenções léxicas das sintáticas pode levar a um projeto global de linguagem mais claro.
2. A eficiência do compilador é melhorada. Um analisador léxico separado nos permite construir um processador especializado e potencialmente mais eficiente para a tarefa. Uma grande quantidade de tempo é gasta lendo-se o programa-fonte e particionando-o em

---

As expressões regulares foram citadas acima para a descrição de padrões. Os termos Lex e Compilador Lex são usados para descrever ferramentas de software que localizam os tokens de uma linguagem de programação com base em uma especificação regular. Veja a Seção 3.5 para mais detalhes.

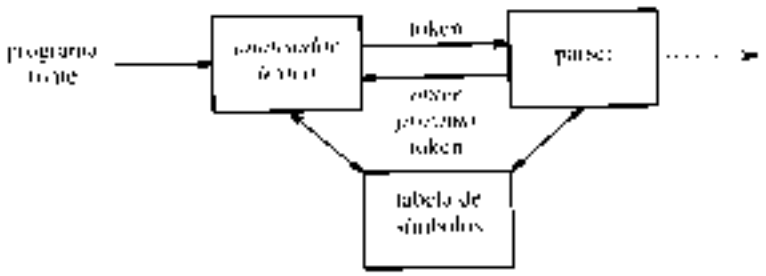


Fig. 3.1. Interações analisador léxico-parser

podem. Técnicas de bufferização especializadas para a leitura de caracteres e o processamento de tokens podem acelerar significativamente o desempenho de um compilador.

- A portabilidade do compilador é realçada. As peculiaridades do alfabeto de entrada e outras anomalias específicas de dispositivos podem ser restringidas ao analisador léxico. A representação de símbolos especiais ou não-padrão, tais como { ou Pascal, pode ser tratada no analisador léxico.

Ferramentas especializadas têm sido projetadas para auxiliar a automação da construção de analisadores léxicos e parsers quando os mesmos estão separados. Veremos exemplos globais de tais ferramentas neste livro.

Tokens, Padrões, Lexemas

Quando se fala sobre análise lexical, existem os termos "token", "padrão" e "lexema" com significados específicos. Exemplos de seus usos são mostrados na Fig. 3.2. Em geral, existe um conjunto de cadeias de entrada para as quais o mesmo *token* é produzido como saída. Esse conjunto de cadeias é descrito por uma regra chamada de um *padrão* associado ao token de entrada. O padrão é dito *combinar* cada cadeia do conjunto. Um lexema é uma sequência de caracteres no programa fonte que é reconhecida pelo padrão de algum token. Por exemplo, na instrução Pascal

```
                    
```

a subcadeia [illegible] é um lexema para o token "identificador".

Tratamos os *tokens* como símbolos terminais na gramática para a linguagem-fonte, usando nomes em negrito para representá-los. Os lexemas reconhecidos pelo padrão do token representam cadeias de caracteres no programa-fonte e podem receber tratamento conjunto, como constantes de uma mesma unidade lexical, por exemplo, instâncias de identificadores, quaisquer que [illegible].

Na maioria das linguagens de programação, as seguintes construções são tratadas como tokens: palavras-chave, operadores, identificadores, constantes, literais, cadeias e símbolos de pontuação, como vírgulas, pontos-e-vírgulas e parênteses. No exemplo acima, quando a sequência de caracteres [illegible] aparece no programa-fonte, um token representando um identificador é repassado ao parser. O repasse de um token é frequentemente implementado passando-se um ponteiro para a tabela de símbolos. É justamente esse interno que é designado por id na Fig. 3.2.

Um padrão é uma regra que descreve o conjunto de lexemas que podem representar um token particular nos programas fonte. O padrão para o token const na Fig. 3.2 é exatamente a cadeia simples [illegible] que soletra a palavra-chave. O padrão para o token **relação** é o conjunto de todos os seis operadores relacionais de Pascal. Para descrever precisamente os padrões para tokens mais complexos, como **id** (identificador) e **num** (número), nós usamos a notação de expressões regulares desenvolvida na Seção 3.3.

Certas convenções de linguagem aumentam a dificuldade da análise léxica. Linguagens como Fortran requerem certas construções em posições fixas na linha de entrada. Dessa forma, o alinhamento de um lexema pode ser importante na determinação da correção de um programa fonte. A tendência nos projetos modernos de linguagens de programação está na direção de entradas em formato livre, permitindo que as construções apareçam em qualquer local de linha de entrada. Uma consequência desse aspecto da análise léxica é ser normalmente mais importante.

O tratamento dos espaços varia grandemente de linguagem para linguagem. Em algumas linguagens, como Fortran e Algol 68, os espaços não são significativos, exceto quando dentro de literais ou cadeias de caracteres. Podem ser adicionados à vontade a fim de melhorar a legibilidade de um programa. As convenções relacionadas aos espaços podem complicar grandemente a tarefa de identificação dos *tokens*.

Um exemplo popular que ilustra a dificuldade potencial em se reconhecer *tokens* é o comando [illegible] de Fortran. No comando

```
                
```

não podemos afirmar que [illegible] seja parte do identificador [illegible] até que tenhamos examinado o ponto decimal. Por outro lado, no comando

```
                
```

TOKEN	LEXEMAS EXEMPLO	DESCRIÇÃO INFORMAL DO PADRÃO
const	[illegible]	[illegible]
if	[illegible]	[illegible]
relação	[illegible]	[illegible]
id	[illegible]	letra seguida por letras e dígitos
num	[illegible]	qualquer constante numérica
literal	[illegible]	qualquer caracteres entre aspas exceto aspas

Fig. 3.2. Exemplos de tokens

tem-se seis *tokens*: a palavra-chave DO, o rótulo de enunciado 5, o identificador I, o operador =, a constante 1, a vírgula e a constante 25. Aqui não podemos estar seguros, até que tenhamos examinado a vírgula, de que DO seja uma palavra-chave. Para aliviar essa incerteza, Fortran 77 permite que uma vírgula opcional seja colocada entre o rótulo e o índice do enunciado DO. No exemplo, DO 5, I=1,25. O uso dessa vírgula é encorajado porque a mesma ajuda a tornar o enunciado DO mais claro e legível.

Em muitas linguagens, certas cadeias são reservadas, isto é, seus significados são pré-definidos e não podem ser modificados pelo usuário. Se uma palavra-chave não for reservada, o analisador léxico precisa distinguir uma palavra-chave de um identificador definido pelo usuário. Em PL/I, as palavras-chave não são reservadas; consequentemente, as regras para essa distinção são um tanto complicadas, como o seguinte enunciado PL/I ilustra:

```
IF THEN THEN THEN = ELSE; ELSE ELSE = THEN;
```

Atributos para os *Tokens*

Quando um lexema for reconhecido por mais de um padrão, o analisador léxico precisa providenciar informações adicionais para as fases subseqüentes do compilador a respeito do lexema particular que foi reconhecido. Por exemplo, o padrão **num** reconhece as duas cadeias 0 e 1, mas é essencial para o gerador de código ser informado sobre que cadeia foi efetivamente reconhecida.

O analisador léxico coleta informações a respeito dos *tokens* em seus atributos associados. Os *tokens* influenciam decisões na análise gramatical; os atributos influenciam a tradução dos *tokens*. Do ponto de vista prático, o *token* possui usualmente somente um único atributo — um apontador para a entrada da tabela de símbolos na qual as informações sobre o mesmo são mantidas; o apontador se torna o atributo do *token*. Para fins de diagnóstico, podemos estar interessados tanto no lexema de um identificador quanto no número da linha na qual o mesmo foi primeiramente examinado. Esses dois itens de informação podem, ambos, ser armazenados na entrada da tabela de símbolos para o identificador.

Exemplo 3.1. Os *tokens* e os valores de atributos associados ao enunciado Fortran

```
E = M * C ** 2
```

são escritos abaixo como uma seqüência de pares:

<**id**, apontador para a entrada da tabela de símbolos para E>

<**operador_de_atribuição**, >

<**id**, apontador para a entrada da tabela de símbolos para M>

<**operador_de_multiplicação**, >

<**id**, apontador para a entrada da tabela de símbolos para C>

<**operador_de_exponenciação**, >

<**num**, valor inteiro 2>

Note-se que em certos pares não existe a necessidade de um valor de atributo; o primeiro componente é suficiente para identificar o lexema. Nesse pequeno exemplo, ao *token* **num** foi associado um atributo de valor inteiro. O compilador pode armazenar a cadeia de caracteres que forma o número numa tabela de símbolos e deixar o atributo do *token* **num** ser o apontador para a entrada da tabela. □

Erros Léxicos

Poucos erros são distinguíveis somente no nível léxico, uma vez que um analisador léxico possui uma visão muito local do programa fonte. Se a cadeia fi for encontrada pela primeira vez num programa C no contexto

```
fi ( a == f(x) ) ...
```

um analisador léxico não poderá dizer se fi é a palavra-chave if incorretamente grafada ou um identificador de função não declarada. Como fi é um identificador válido, o analisador léxico precisa retornar o *token* identificador e deixar alguma fase posterior do compilador tratar o eventual erro.

Mas, suponhamos que surja uma situação na qual o analisador léxico seja incapaz de prosseguir, porque nenhum dos padrões reconheça um prefixo da entrada remanescente. Talvez a estratégia mais simples de recuperação seja a da "modalidade pânico". Removemos sucessivos caracteres da entrada remanescente até que o analisador léxico possa encontrar um *token* bem formado. Essa técnica de recuperação pode ocasionalmente confundir o *parser*, mas num ambiente de computação interativo pode ser razoavelmente adequada.

Outras possíveis ações de recuperação de erros são:

1. remover um caractere estranho
2. inserir um caractere ausente
3. substituir um caractere incorreto por um correto
4. transpor dois caracteres adjacentes

Transformações de erros como essas podem ser experimentadas numa tentativa de se consertar a entrada. A mais simples de tais estratégias é a de se verificar se um prefixo da entrada remanescente pode ser transformado num lexema válido através de uma única transformação. Essa estratégia assume que a maioria dos erros léxicos seja resultante de uma única transformação de erro, uma suposição usualmente confirmada na prática, embora nem sempre.

Uma forma de se encontrar erros num programa é computar o número mínimo de transformações de erros requeridas para tornar um programa errado num que seja sintaticamente bem-formado. Dizemos que um programa errado possui *k* erros se a menor seqüência de transformações de erros que irá mapeá-lo em algum programa válido possui comprimento *k*. A correção de erros de distância mínima é uma conveniente ferramenta teórica de longo alcance, mas que não é geralmente usada por ser custosa demais de implementar. Entretanto, uns poucos compiladores experimentais têm usado o critério da distância mínima para realizar correções localizadas.

3.2 BUFERIZAÇÃO DA ENTRADA

Esta seção cobre alguns temas de desempenho relacionados à bufferização da entrada. Mencionamos primeiramente um esquema de entrada com dois *buffers*, que é útil quando uma pré-varredura da entrada é necessária para identificar os *tokens*. Introduzimos, então, algumas técnicas úteis para acelerar o analisador léxico, tais como o uso de "sentinelas" para marcar o final do *buffer*.

Existem três enfoques gerais para a implementação de um analisador léxico:

1. Usar um gerador de analisadores léxicos, tal como o compilador Lex, discutido na Seção 3.5, a fim de produzir o analisador léxico a partir de uma especificação baseada em expressões regulares. Nesse caso, o gerador providencia as rotinas para a leitura e a bufferização da entrada.
2. Escrever um analisador léxico numa linguagem de programação de sistemas convencional, usando as facilidades de E/S da mesma para ler a entrada.

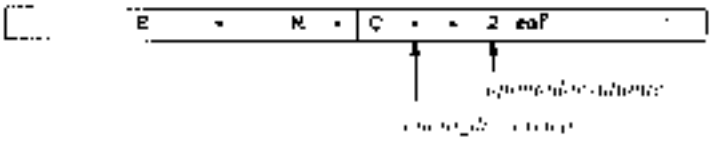


Fig. 3.3: Um *buffer* com seus dois apontadores.

3. Escreva um analisador léxico em uma linguagem de montagem e mantenha explicitamente o estado do autômato.

Se todas essas alternativas parecem muito complexas, não se preocupe. Existem ferramentas que fazem o trabalho pesado para você. Nos capítulos 2 e 4 vamos ver como usar o Lex e o Flex, duas ferramentas muito populares para gerar analisadores léxicos. Mesmo se você for escrever um analisador léxico à mão, é importante entender como essas ferramentas funcionam internamente. Muitos dos erros cometidos por programadores iniciantes ao escrever analisadores léxicos podem ser facilmente evitados se você souber como a entrada é bufferizada.

Pares de *Buffers*

Muitas linguagens de programação permitem que o programador defina novos operadores. Por exemplo, em Pascal, se quisermos definir o operador `**` para representar a exponenciação, podemos escrever uma função `operator **(int, int)` e usar `a ** b` em vez de `exponencial(a, b)`. Em Prolog, podemos definir `**` como um operador infixo com `op(500, xfy, **)` e escrever `a ** b` em vez de `**(a, b)`. Em ambos os casos, o analisador sintático precisa ser informado sobre a existência do novo operador antes de encontrá-lo em um programa. Em Pascal, isso é feito através de uma declaração `operator`; em Prolog, através de uma chamada à função `op`. Em ambas as linguagens, a declaração deve preceder o uso do operador.

Uma exceção a essa regra ocorre quando o operador é definido em termos de si mesmo. Por exemplo, se estivermos escrevendo um interpretador Prolog em Prolog, precisaremos ler a definição de `op` antes de sabermos que `op` é um operador. Para evitar esse problema, o interpretador deve ser escrito em uma linguagem que permita a definição de novos operadores sem que eles sejam declarados previamente.

Uma maneira de implementar operadores definidos pelo usuário é através de uma tabela de símbolos. Quando o operador é definido, uma entrada é criada na tabela de símbolos para armazenar suas propriedades. Quando o operador é encontrado em uma expressão, o analisador sintático consulta a tabela de símbolos para descobrir como ele deve ser tratado. Essa abordagem é simples e flexível, mas também é lenta. Cada operador encontrado durante a análise sintática requer uma consulta à tabela de símbolos.

Uma abordagem mais eficiente é gerar código para o analisador sintático que reconhece as expressões envolvendo os operadores definidos pelo usuário. Em Prolog, por exemplo, o código gerado para `a ** b` pode ser algo como `**(a, b)`. Em Pascal, o código gerado para `a ** b` pode ser algo como `exponencial(a, b)`. Em ambos os casos, o analisador sintático não precisa consultar a tabela de símbolos para descobrir como tratar o operador `**`. Em vez disso, ele simplesmente gera o código apropriado.

Essa abordagem é muito eficiente. Infelizmente, ela também é muito complexa. Felizmente, as abordagens mais difíceis são simplesmente produzem frequentemente os analisadores léxicos mais rápidos. Como o analisador léxico é a fase do compilador que lê o programa-fonte caractere a caractere, é possível ganhar uma quantidade considerável de tempo tornando o analisador léxico o mais rápido possível. Em geral, os analisadores sintáticos são muito mais complexos. Consequentemente, é difícil o analisador sintático ser o gargalo de um compilador.

Existem várias técnicas para melhorar a velocidade de um analisador léxico. Vamos discutir duas delas: *pares de buffers* e *sentinelas*. Ambas as técnicas são projetadas para reduzir o número de instruções executadas para ler um caractere de entrada. Em vez de ler um caractere por vez, o analisador léxico lê blocos de caracteres de uma só vez. Isso é feito através de uma chamada de sistema, como `read` em UNIX ou `BlockRead` em Turbo Pascal. Se você não tiver acesso a essas chamadas de sistema, pode usar `getchar` ou `getc` em C, ou `get` em Pascal. Mesmo assim, você deve ler a entrada em blocos, e não um caractere por vez.

Um par de *buffers* é um vetor de caracteres dividido em duas metades, cada uma com N caracteres, como mostrado na Fig. 3.3. Tipicamente, N é igual ao número de caracteres em um bloco de disco, por exemplo, 1024 ou 4096.

Em vez de ler caracteres de entrada um de cada vez, lemos N caracteres de uma só vez com uma única chamada de sistema. Se houver menos de N caracteres de entrada antes do fim do arquivo, um caractere especial, `eof`, é armazenado no buffer. Esse caractere especial não pode ser confundido com nenhum caractere válido do programa-fonte. Uma escolha natural é `EOF`. A Fig. 3.3 mostra um par de *buffers* que contém os lexemas `if` e `then`.

Como ilustrado na Fig. 3.3, podemos usar o código mostrado na Fig. 3.4 para avançar o apontador adiante de cada caractere lido. O código realiza um teste para ver se chegamos ao fim de uma metade do buffer ou ao fim do arquivo. Se nenhum dos dois for o caso, o apontador é simplesmente incrementado. Caso contrário, um teste adicional é feito para ver se o fim da primeira metade foi encontrado. Se for o caso, a segunda metade é recarregada. Caso contrário, o fim do arquivo foi encontrado e a análise léxica pode ser concluída.

```
se apontador_atual = fim da primeira metade
então início
    recarregar a segunda metade;

apontador_atual := apontador_atual + 1

fim
senão se apontador_atual = fim da segunda metade
então início
    recarregar a primeira metade;
    deslocar apontador_atual para o início da primeira
    metade

fim
senão apontador_atual := apontador_atual + 1;
```

Fig. 3.4: Código para avançar o apontador de entrada.

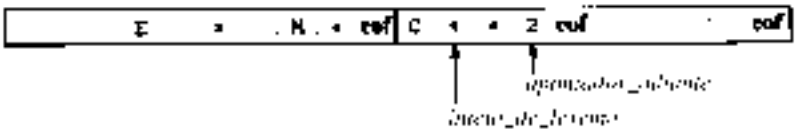


Fig. 3.5. Sentinelas em cada metade do buffer.

Também precisamos decidir como processar os caracteres especiais delimitadores pelo apontador adiante (marca o final do token, representa uma das quatro possibilidades para o final de uma palavra-chave particular, por exemplo). Uma forma de testar esses testes é usar um enumerado case, se a implementação da linguagem possuir um. O teste

se *apontador_adiante*^ = eof

pode ser implementado como uma das diferentes alternativas de um comando case.

3.3 ESPECIFICAÇÃO DOS *TOKENS*

As expressões regulares são uma notação importante para especificar padrões. Cada padrão corresponde a um conjunto de cadeias e, dessa forma, as expressões regulares serão muito convenientes para conjuntos de cadeias. A Seção 3.5 estende esta notação numa linguagem orientada a padrões para a análise léxica.

Cadeias e Linguagens

O termo *alfabeto* ou *classe de caracteres* denota qualquer conjunto finito de símbolos. Exemplos típicos de símbolos são letras e caracteres. O conjunto {0, 1} é o alfabeto binário. EBCDIC e ASCII são dois exemplos de alfabetos de computadores.

Uma *cadeia* sobre algum alfabeto é uma sequência finita de símbolos retirados do mesmo. Na teoria das linguagens, os termos *sentença* e *palavra* são frequentemente usados como sinônimos para o termo cadeia. O comprimento da cadeia *s* usualmente é escrito |*s*| e é o número de ocorrências de símbolos em *s*. Por exemplo, banana é uma cadeia de comprimento seis. A cadeia vazia, denotada ε, é uma cadeia especial de comprimento zero. Alguns termos comuns, associados a partes de uma cadeia, estão sumarizados na Fig. 3.7.

O termo *linguagem* denota qualquer conjunto de cadeias sobre algum alfabeto fixo. A definição é muito ampla. Linguagens abstratas como ∅, o conjunto vazio, ou {ε}, o conjunto contendo somente a cadeia vazia, são linguagens sob essa definição. Também o conjunto de todos os programas Pascal sintaticamente bem formados é uma linguagem, embora o conjunto de todas as sentenças gramaticalmente corretas em inglês, apesar de também ser uma linguagem, seja muito difícil de se especificar. Note-se também que essa definição não preserva nenhum significado para as cadeias de uma linguagem. Os métodos para a preservação de significados são discutidos no Capítulo 8.

Se *x* e *y* são cadeias, então a concatenação de *x* e *y*, escrita como *xy*, é a cadeia formada pela cadeia *x* seguida pela cadeia *y*. Por exemplo, se *x* = dog e *y* = house, então *xy* = doghouse. A cadeia vazia é o elemento identidade da concatenação, isto é, *sε* = *εs* = *s*.

Se pensarmos na concatenação como um "produto", podemos definir a cadeia "exponenciação" assim segue. Definir s^0 como sendo ε e, para $i > 0$, definir s^i como sendo $s^{i-1}s$. Como a cadeia *s* é *s*ε, $s^1 = s$. Dessa forma, $s^2 = ss$, $s^3 = sss$ e assim por diante.

Operações em Linguagens

Existem várias operações importantes que podem ser aplicadas às linguagens. Para a análise léxica, estaremos principalmente interessados em união, concatenação e fechamento, as quais são definidas na Fig. 3.8. Podemos também generalizar o operador de "exponenciação" para linguagens, definindo L^0 como sendo {ε} e L^i como $L^{i-1}L$. Dessa forma, L^2 é L concatenada consigo mesma duas vezes.

Exemplo 3.2. Seja *L* o conjunto {A, B, ..., Z, a, b, ..., z} e *D* o conjunto {0, 1, ..., 9}. Podemos pensar de *L* e *D* em duas formas. *L* pode ser o alfabeto que consiste no conjunto de letras maiúsculas e minúsculas e *D* o dos dez dígitos decimais. Alternativa-

TERMO	DEFINIÇÃO
prefixo de *s*	Uma cadeia obtida pela remoção de zero ou mais símbolos ao fim da cadeia *s*; por exemplo, ban é um prefixo de banana.
sufixo de *s*	Uma cadeia obtida pela remoção de zero ou mais símbolos ao início da cadeia *s*; por exemplo, nana é um sufixo de banana.
subcadeia de *s*	Uma cadeia obtida pela remoção de um prefixo e de um sufixo de *s*; por exemplo, nan é uma subcadeia de banana. Cada prefixo e cada sufixo de *s* é uma subcadeia de *s*, mas nem toda subcadeia de *s* é um prefixo ou um sufixo de *s*. Para cada cadeia *s*, tanto *s* quanto ε são prefixos, sufixos e subcadeias de *s*.
prefixo, sufixo ou subcadeia própria de *s*	Qualquer cadeia não vazia *x* que seja, respectivamente, prefixo, sufixo ou subcadeia de *s* tal que $s \neq x$.
subsequência de *s*	Qualquer cadeia formada pela remoção de zero ou mais símbolos não necessariamente contíguos de *s*; por exemplo, baaa é uma subsequência de banana.

Fig. 3.7. Termos para partes de uma cadeia.

```
apontador_adiante := apontador_adiante + 1;
se apontador_adiante^ = eof
então início
        se apontador_adiante estiver ao fim da primeira metade
        então início
                recarregar segunda metade;
                apontador_adiante := apontador_adiante + 1
        fim
        senão se apontador_adiante estiver ao fim da segunda
        metade
        então início
                recarregar primeira metade;
                mover apontador_adiante para o início da
                primeira metade
        fim
        senão { eof está na borda} indicando fim da entrada }
                terminar a análise léxica
fim
```

Fig. 3.6. Código de estado finito para uma especificação com sentinelas.

Operação	Definição
união de L e M, escrita $L \cup M$	$L \cup M = \{s \mid s \text{ está em } L \text{ ou está em } M\}$
concatenação de L e M, escrita LM	$LM = \{st \mid s \text{ está em } L \text{ e } t \text{ está em } M\}$
fecho de Kleene de L, escrito L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ $L^0 = \{\varepsilon\}$
L elevado k	L^k denota "zero ou mais concatenações de" L.
fecho positivo de L, escrito L^+	$L^+ = \bigcup_{i=1}^{\infty} L^i$ $L^1 = L$
L elevado k	L^k denota uma ou mais concatenações de L

Fig. 3.8. Definições de operações em linguagens.

tegia como no símbolo pode ser considerado ou a cadeia de comprimento um, os conjuntos L e D são, cada um, linguagens finitas. Aqui estão alguns exemplos de novas linguagens criadas a partir de L e D, através da aplicação dos operadores definidos na Fig. 3.8:

1. $L \cup D$ é o conjunto de letras e dígitos.
2. LD é o conjunto de cadeias constituídas de uma letra seguida por um dígito.
3. L^4 é o conjunto de todas as cadeias com quatro letras.
4. L^* é o conjunto de todas as cadeias de letras, incluindo a cadeia vazia.
5. $L(L \cup D)^*$ é o conjunto de todas as cadeias de letras e dígitos que começam por uma letra.
6. D^+ é o conjunto de todas as cadeias de um ou mais dígitos.

Expressões Regulares

Em Pascal, um identificador é uma letra seguida por zero ou mais letras ou dígitos. Já vimos um identificador como membro do conjunto definido na parte (5) do Exemplo 3.2. Nesta seção apresentaremos uma notação, chamada de expressões regulares, que nos permite definir precisamente conjuntos daquela natureza. Com essa notação, definiríamos identificadores Pascal como

letra (letra | dígito)*

Aqui a barra vertical significa "ou", os parênteses são usados para agrupar subexpressões, e o asterisco significa "zero ou mais instâncias" da expressão parentetizada. A justaposição de letra com o resto da expressão significa concatenação.

Uma expressão regular é construída de expressões regulares mais simples através de um conjunto de regras de definição. Cada expressão regular r denota uma linguagem $L(r)$. As regras de definição especificam como $L(r)$ é formada através da combinação, em certas formas, das linguagens denotadas pelas subexpressões de r.

A seguir estão as regras que definem as *expressões regulares* sobre um alfabeto Σ. Associada a cada regra existe uma especificação da linguagem denotada pela expressão regular sendo definida.

1. ε é uma expressão regular que denota $\{\varepsilon\}$, isto é, o conjunto que contém a cadeia vazia.
2. Se a é um símbolo em Σ, então a é uma expressão regular que denota $\{a\}$, isto é, o conjunto contendo a cadeia a. Apesar de as expressões a mesma notação para ambos os fins, fica claro pelo contexto se estamos falando sobre a expressão regular a ou sobre o símbolo a. Para mais clareza a partir do contexto se estamos falando sobre a expressão regular, cadeia ou símbolo.
3. Suponhamos que r e s sejam expressões regulares denotando as linguagens $L(r)$ e $L(s)$. Dessa forma,
   a) $(r)|(s)$ é uma expressão regular denotando $L(r) \cup L(s)$.
   b) $(r)(s)$ é uma expressão regular denotando $L(r)L(s)$.
   c) $(r)^*$ é uma expressão regular denotando $(L(r))^*$.
   d) (r) é uma expressão regular denotando $L(r)$.

A linguagem definida por uma expressão regular é dita ser um *conjunto regular*.

A especificação de uma expressão regular é um exemplo de definição recursiva. As regras (1) e (2) formam a base da definição e vamos dizer que *símbolo básico* é um de nos elementos de ε ou um símbolo em Σ figurando numa expressão regular. A regra (3) provê o passo de indução.

Os parênteses desnecessários podem ser evitados nas expressões regulares se adotarmos as convenções em que:

1. o operador unário $*$ possui a mais alta precedência e seja associativo à esquerda;
2. a concatenação tenha a segunda mais alta precedência e seja associativa à esquerda;
3. $|$ possua a mais baixa precedência e seja associativo à esquerda.

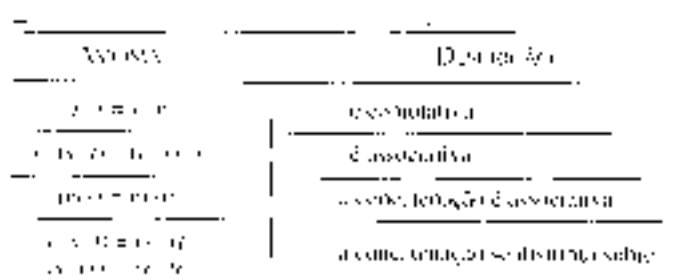
Sob essas convenções $(r)|(s)$ é o mesmo que $r|s$. Ambos os expressões denotam conjuntos de cadeias que sejam uma letra ou zero ou mais letras ou dígitos, segundo a primeira.

Exemplo 3.3. Seja $\Sigma = \{a, b\}$.

1. A expressão regular $a|b$ denota o conjunto $\{a, b\}$.
2. A expressão regular $(a|b)(a|b)$ denota $\{aa, ab, ba, bb\}$, o conjunto de todas as cadeias de a's e b's de comprimento dois. Outra expressão regular para esse mesmo conjunto é $aa|ab|ba|bb$.
3. A expressão regular a^* denota o conjunto de todas as cadeias de zero ou mais a's, isto é, $\{\varepsilon, a, aa, aaa, \ldots\}$.
4. A expressão regular $(a|b)^*$ denota o conjunto de todas as cadeias contendo zero ou mais instâncias de a ou b, ou seja, o conjunto

---

[1] Nesta e em muitas outras ocasiões, parênteses podem ser colocados em uma expressão regular sem modificá-la.



Axioma	Descrição
$r \mid s = s \mid r$	$\mid$ é comutativa
$r \mid (s \mid t) = (r \mid s) \mid t$	$\mid$ é associativa
$r(st) = (rs)t$	concatenação é associativa
$r(s \mid t) = rs \mid rt$ $(s \mid t)r = sr \mid tr$	concatenação é distributiva sobre $\mid$
$\varepsilon r = r$ $r\varepsilon = r$	ε é o elemento identidade da concatenação
$r^* = (r \mid \varepsilon)^*$	relação entre $*$ e ε
$r^{**} = r^*$	$*$ é idempotente

Fig. 3.9. Propriedades algébricas de expressões regulares.

ponto de todas as cadeias desse tipo. Outra expressão regular para o mesmo conjunto é $(0 \mid 1)^*1$.

5. A expressão regular $[A\text{-}Za\text{-}z][A\text{-}Za\text{-}z0\text{-}9]^*$ denota o conjunto contendo todas as letras e todas as cadeias consistindo em zero ou mais letras e dígitos seguindo pelo nome. ∎

Se duas expressões regulares r e s denotam a mesma linguagem, podemos escrever $r = s$. Por exemplo, $1 \mid 0 = 0 \mid 1$.

Existe um número de leis algébricas válidas para expressões regulares e que podem ser usadas para manipulá-las em formas equivalentes. A Figura 3.9 mostra algumas leis algébricas que se aplicam a expressões regulares r, s e t.

Definições Regulares

Por conveniência de notação, podemos dar nomes a expressões regulares e definir expressões regulares usando esses nomes como se fossem símbolos. Se Σ for um alfabeto de símbolos básicos, então uma *definição regular* é uma sequência de definições da forma

$$\begin{aligned} d_1 &\to r_1 \\ d_2 &\to r_2 \\ &\cdots \\ d_n &\to r_n \end{aligned}$$

onde cada d_i é um novo símbolo distinto e cada r_i é uma expressão sobre os símbolos em $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$, isto é, os símbolos básicos e os nomes previamente definidos. Restringindo cada r_i a símbolos de Σ e aos nomes previamente definidos, podemos construir, uma expressão regular sobre Σ para qualquer r_i substituindo sucessivamente os nomes de expressões regulares pelas expressões que esses nomes denotam. Se r_i usasse, por engano, d_i ou d_j para $j > i$, então essa substituição poderia não terminar, pois estaríamos definindo d_i em termos de si mesmo.

A fim de distinguir os nomes dos símbolos, imprimiremos os nomes nas definições regulares em negrito.

Exemplo 3.4. Como nos antigos estabelecedores de conjuntos de identificadores Pascal é o conjunto de cadeias de letras e dígitos começando por uma letra. A seguinte sequência de definição regular para esse conjunto

$$\begin{aligned} \textbf{letra} &\to \mathtt{A} \mid \mathtt{B} \mid \cdots \mid \mathtt{Z} \mid \mathtt{a} \mid \mathtt{b} \mid \cdots \mid \mathtt{z} \\ \textbf{digito} &\to \mathtt{0} \mid \mathtt{1} \mid \cdots \mid \mathtt{9} \\ \textbf{id} &\to \textbf{letra} \; (\; \textbf{letra} \mid \textbf{digito} \;)^* \end{aligned}$$

Exemplo 3.5. Os números sem sinal em Pascal são cadeias tais como [illegible]. A seguinte definição regular pode ser usada como parte de uma precisa especificação para esse conjunto de cadeias:

$$\begin{aligned} \textbf{digito} &\to \mathtt{0} \mid \mathtt{1} \mid \cdots \mid \mathtt{9} \\ \textbf{digitos} &\to \textbf{digito} \; \textbf{digito}^* \\ \textbf{fração\_opcional} &\to (\; \mathtt{.} \; \textbf{digitos} \;) \mid \varepsilon \\ \textbf{expoente\_opcional} &\to (\; \mathtt{E} \; (\; \mathtt{+} \mid \mathtt{-} \mid \varepsilon \;) \; \textbf{digitos} \;) \mid \varepsilon \\ \textbf{num} &\to \textbf{digitos} \; \textbf{fração\_opcional} \; \textbf{expoente\_opcional} \end{aligned}$$

Esta definição diz que uma **fração_opcional** pode ser um ponto decimal seguido por uma ou mais dígitos, ou pode ser ausente (a cadeia vazia). Um **expoente_opcional**, se não estiver ausente, é um E seguido por um $+$, $-$ ou nada, seguido por um ou mais dígitos. Observe que podemos usar a definição regular de **digitos** para descrever a sequência de um ou mais dígitos, mas não podemos usar [illegible] para descrever [illegible]. ∎

Simplificações Notacionais

As duas convenções a seguir podem ser frequentemente usadas em expressões regulares que é conveniente introduzir algumas simplificações notacionais para as mesmas.

1. *Um ou mais exemplares.* O operador unário pós-fixo $+$ significa "um ou mais exemplares de". Se r for uma expressão regular que denota a linguagem $L(r)$, então $(r)^+$ é uma expressão regular que denota a linguagem $(L(r))^+$. Dessa forma a expressão regular a^+ denota o conjunto de todas as cadeias de um ou mais a's. O operador $+$ possui a mesma precedência e associatividade que o operador $*$. As duas identidades algébricas $r^* = r^+ \mid \varepsilon$ e $r = r^+ \mid \varepsilon$ relacionam os operadores de fecho, incluindo Kleene e positivo.
2. *Zero ou uma ocorrência.* O operador pós-fixo unário $?$ significa "zero ou uma ocorrência de". A notação $r?$ é uma simplificação para $r \mid \varepsilon$. Se for uma expressão regular, então $(r)?$ denota a linguagem $L(r) \cup \{\varepsilon\}$. Por exemplo, usando os operadores $+$ e $?$, podemos reescrever a definição regular de **num** no Exemplo 3.5 como

$$\begin{aligned} \textbf{digito} &\to \mathtt{0} \mid \mathtt{1} \mid \cdots \mid \mathtt{9} \\ \textbf{digitos} &\to \textbf{digito}^+ \\ \textbf{fração\_opcional} &\to (\; \mathtt{.} \; \textbf{digitos} \;)? \\ \textbf{expoente\_opcional} &\to (\; \mathtt{E} \; (\; \mathtt{+} \mid \mathtt{-} \;)? \; \textbf{digitos} \;)? \\ \textbf{num} &\to \textbf{digitos} \; \textbf{fração\_opcional} \; \textbf{expoente\_opcional} \end{aligned}$$

3. *Classes de caracteres.* A notação $[abc]$, onde a, b e c são símbolos básicos, é uma abreviação para a expressão regular $a \mid b \mid c$. Uma classe de caracteres abreviada tal como $[a\text{-}z]$ denota a expressão regular $a \mid b \mid \cdots \mid z$. Usando essas classes de caracteres, podemos reescrever a definição de **id** como sendo cadeias geradas pela expressão regular

$$[\mathtt{A\text{-}Za\text{-}z}]\,[\mathtt{A\text{-}Za\text{-}z0\text{-}9}]^*$$

Conjuntos Não-regulares

Algumas linguagens não podem ser descritas por qualquer expressão regular. A fim de ilustrar os limites do poder descritivo das expressões regulares, forneceremos, neste ponto, exemplos de construções de linguagens de programação que não podem ser descritas por expressões regulares. As provas dessas assertivas podem ser encontradas nas referências.

As expressões regulares não podem ser usadas para descrever parênteses balanceados (ou construções aninhadas). Por exemplo, o conjunto de todas as cadeias de parênteses balanceados não pode ser des-

cuja parte direita é uma expressão regular. Portanto, todo o conjunto pode ser descrito por uma gramática livre de contexto.

As classes de *tokens* que podem ser descritas pelas expressões regulares do conjunto

$$\{ws, if, then, else, id, num, relop\}$$

não pode ser descritas por qualquer expressão regular, mas pode ser descrita por uma gramática livre de contexto.

As expressões regulares podem ser usadas para descrever somente um número limitado de repetições ou um número não especificado de repetições de mesma natureza (consulte os Exercícios 3.7 e 3.8). Assim, os identificadores não podem ser encapsulados dentro de expressões regulares nos Pascals da vida. Nessa linha de raciocínio, não podemos descrever um par de parênteses balanceados. Dessa forma, não podemos descrever uma expressão regular que reconheça corretamente todas as construções aritméticas bem formadas. Como as expressões regulares são capazes de descrever, porém, o número de construções sintáticas encontradas na prática, geralmente, é suficiente, o número de situações de [illegible].

3.4 O RECONHECIMENTO DE *TOKENS*

Nesta seção, consideraremos o problema de como reconhecer os *tokens* descritos por expressões regulares. Nesta seção, estudaremos a questão de como reconhecer os *tokens*. Ao longo desta seção, usaremos uma linguagem gerada pela gramática seguinte como um exemplo em execução.

Exemplo 3.6. Considere a seguinte fragmento de gramática:

$$\begin{aligned} stmt &\rightarrow \textbf{if } expr \textbf{ then } stmt \\ &\mid \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ &\mid \varepsilon \\ expr &\rightarrow term \textbf{ relop } term \\ &\mid term \\ term &\rightarrow \textbf{id} \\ &\mid \textbf{num} \end{aligned}$$

onde os termos em negrito **if**, **then**, **else**, **relop**, **id** e **num** geram conjuntos de cadeias dados pelas seguintes definições regulares:

$$\begin{aligned} \textbf{if} &\rightarrow \text{if} \\ \textbf{then} &\rightarrow \text{then} \\ \textbf{else} &\rightarrow \text{else} \\ \textbf{relop} &\rightarrow < \mid <= \mid = \mid <> \mid > \mid >= \\ \textbf{id} &\rightarrow \textbf{letra} \; (\textbf{letra} \mid \textbf{digito})^* \\ \textbf{num} &\rightarrow \textbf{digito}^+ \; (. \; \textbf{digito}^+)? \; (\text{E} \; (+ \mid -)? \; \textbf{digito}^+)? \end{aligned}$$

onde **letra** e **digito** são como definidos anteriormente.

Para ver como fragmentos de linguagem como o anterior levam a reconhecedores para suas palavras-chave [illegible], vejamos como os lexemas desses tokens devem ser reconhecidos. Como os lexemas de **relop**, **id** e **num** são arbitrários, as palavras-chave devem ser reservadas, ou seja, não podem ser usadas como identificadores. Como no Exemplo 3.5, **num** representa a forma usual de números reais em Pascal.

Adicionalmente, assumimos que os lexemas são separados por espaços em branco, consistindo em espaços, tabulações e caracteres de linha. Nosso analisador léxico irá remover esses espaços em branco. Definimos a variável sintática **delim** para espaço em branco e definimos **ws** para sequências de espaços em branco:

$$\begin{aligned} \textbf{delim} &\rightarrow \text{branco} \mid \text{tabulação} \mid \text{avanço de linha} \\ \textbf{ws} &\rightarrow \textbf{delim}^+ \end{aligned}$$

Se o reconhecedor encontrar **ws**, ele não retorna ao analisador léxico, mas começa a ler a partir do próximo caractere. Assim, ele prossegue lendo até encontrar um caractere que não seja um espaço em branco e retorna o que for encontrado.

Expressão Regular	Token	Valor de Atributo
ws	–	
if	if	
then	then	–
else	else	
id	id	apontador para a tabela de símbolos
num	num	apontador para a tabela de símbolos
relop	relop	
relop	relop	
relop	relop	
relop	relop	
relop	relop	
relop	relop	

Fig. 3.10. Padrões de expressões regulares para [illegible]

Nesse momento, o reconhecedor terá sido levado a que não [illegible] leia a próxima parte da entrada, e possa [illegible] com um token apropriado. Novamente, o valor de atributo, instalado na tabela de tradução dada na Fig. 3.10, [illegible] para os operadores relacionais são dados pelas constantes simbólicas [illegible].

Diagramas de Transições

Como um passo intermediário na construção de um analisador léxico, produziremos primeiro um fluxograma estilizado, chamado de *diagrama de transição*. Os diagramas de transição descrevem as ações que ocorrem [illegible] quando um analisador léxico é chamado pelo parser a fim de obter o próximo token, como sugerido pela Fig. 3.1. Suponhamos que a entrada de cadeia seja como na Fig. 3.2 e que o apontador de início e de avanço [illegible] caracteres que se seguem ao último lexema encontrado. Usamos um diagrama de transições a fim de controlar o reconhecimento, baseando-se nas informações a respeito dos caracteres que são encontrados na medida em que o apontador avança pela esquerda direita da entrada. Realizamos isso pela [illegible] posição a posição no diagrama. À medida que os caracteres são lidos,

As posições num diagrama de transições são desenhadas como círculos e são chamadas de *estados*. Os estados são conectados por setas, chamadas de *lados*. Os lados que deixam um estado apresentam rótulos indicando os caracteres de entrada que podem aparecer após o diagrama de transições ter atingido o estado. O rótulo *outro* se refere a qualquer caractere que não seja indicado por qualquer um dos lados que deixam o

Assumimos que os diagramas de transições desta seção são determinísticos; isto é, nenhum símbolo pode rotular [illegible] dois lados diferentes que deixam o mesmo estado. Começando na Seção 3.5, relaxaremos essa condição, tornando muito mais simples a vida de um projetista de analisadores léxicos e, com as ferramentas adequadas, não muito difícil para a implementador.

Um estado é rotulado como o estado de *partida*, e é [illegible] o início do diagrama de transições, onde temos o controle quando iniciamos o reconhecimento de um *token*. Certos estados podem ter ações

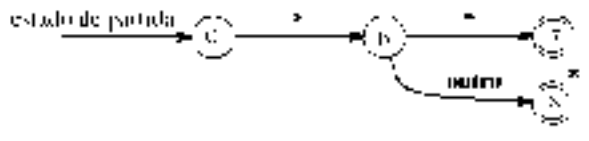


Fig. 3.11. Diagrama de transições [illegible]

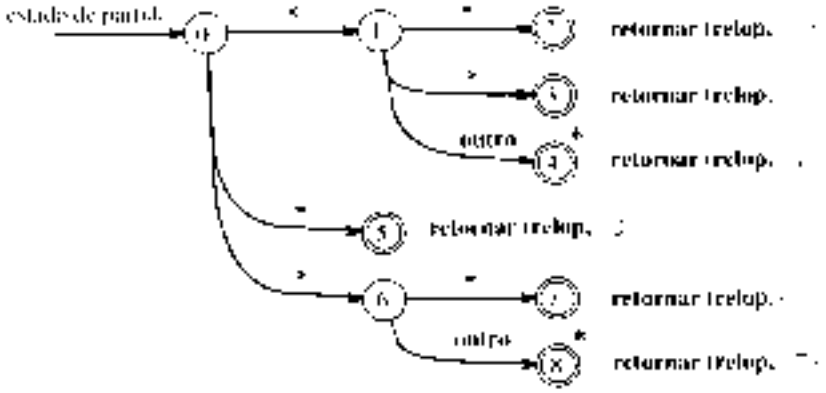


Fig. 3.12. Diagrama de transições para operadores relacionais

que são executados quando atingidos pelo fluxo de controle. Ao entrar num estado, lemos o próximo caractere de entrada. Se existir um lado a partir do estado corrente cujo rótulo corresponde a esse caractere de entrada, avançamos, então, para o estado apontado pelo lado. De maneira análoga, indicamos uma falha.

A Figura 3.11 mostra um diagrama de transições para os padrões `= <> < <= > >=`. O diagrama funciona como se segue: seu estado inicial é o estado 0. No início, lemos o próximo caractere de entrada. O lado rotulado `=` a partir do estado 0 deve ser seguido se o próximo caractere de entrada for `=`. De outra forma, falhamos em reconhecer `=` no estado 0.

Ao atingirmos o estado 0, lemos o próximo caractere de entrada. O lado rotulado `=` a partir do estado 6 deve ser seguido até o estado 7, se o caractere de entrada for `=`. De outra forma, o lado rotulado **outro** indica que devemos nos dirigir ao estado 8. O círculo duplo no estado 7 informa que o mesmo é um estado de aceitação, no qual `>=` foi encontrado.

Note-se que o caractere `>` é um outro caractere extraído durante a medida em que seguimos a sequência de lados a partir do estado de partida até o estado de aceitação 8. Como o caractere extra não é parte do operador relacional `>`, precisamos retrair o apontador adiante em exatamente um caractere. Nesse caso, usamos um `*` para indicar estados nos quais essa ação de retração da entrada precisa tomar lugar.

Em geral, pode haver vários diagramas de transições, cada qual especificando um grupo de *tokens*. Se mantivermos uma falha enquanto estivermos seguindo lados a partir de transições, retraímos o apontador adiante para onde ele mesmo estava quando o estado de partida do diagrama em execução era o próximo diagrama de transições. Os dois apontadores de início de lexema e adiante marcam a mesma posição no estado de partida do diagrama. O apontador adiante é retraído à posição marcada pelo apontador de início de lexema. Se uma falha ocorre em todos os diagramas de transições, um erro léxico foi detectado e invocamos uma rotina de recuperação de erros.

Exemplo 3.7. Um diagrama de transições para o *token* **relop** é mostrado na Fig. 3.12. Note-se que a Fig. 3.11 é uma parte desse diagrama de transições mais complexo.

Exemplo 3.8. Os *tokens* para palavras-chave são sequências de letras, como já vimos. Uma exceção a regra que estabelece que uma sequência de letras e dígitos iniciada por uma letra seja um identificador. Ao invés de codificar as exceções num diagrama de transições, uma saída elegante é a de tratar palavras-chave como identificadores especiais, tal como na Seção 2.7. Quando o estado de aceitação na Fig. 3.13 é atingido, chegamos algum código para determinar se o lexema que levou ao estado de aceitação é uma palavra-chave ou um identificador.

Uma técnica simples para separar as palavras-chave dos identificadores é a de inicializar apropriadamente a tabela de símbolos, onde as informações sobre os identificadores são guardadas. Para os *tokens* da Fig. 3.10, precisamos dar entrada às cadeias `if`, `then`, `else` e `id` na tabela de símbolos antes que quaisquer caracteres de entrada sejam examinados. Também fazemos uma anotação na entrada da tabela de símbolos a respeito do *token* a ser retornado, quando uma dessas cadeias for reconhecida. O enunciado de retorno em seguida ao estado de aceitação na Fig. 3.13 usa *obtem_token* e *instalar_id* para obter, respectivamente, o *token* e o valor de atributo a serem retornados. O procedimento *instalar_id* tem acesso ao *buffer* onde o lexema do identificador foi localizado. A tabela de símbolos é examinada a fim de descobrir se o lexema em questão é uma palavra-chave ou um identificador. Se o lexema for encontrado listalmente como uma palavra-chave, *instalar_id* retorna 0. Se o lexema for encontrado na tabela de símbolos como uma variável de programa, *instalar_id* retorna um apontador para a entrada da tabela de símbolos. Se o lexema não for encontrado na tabela de símbolos, é instalado como uma variável de programa e um apontador para a entrada recentemente criada é retornado.

O procedimento *obter_token* instalarmente procura pelo lexema na tabela de símbolos. Se o lexema for uma palavra-chave, o código correspondente é retornado, de outra forma, o *token* **id** é retornado.

Note-se que o diagrama de transições não precisa ser alterado se palavras-chave adicionais precisarem ser reconhecidas; simplesmente inicializamos a tabela de símbolos com as cadeias e *tokens* das palavras-chave adicionais. ☐

A técnica de se colocar palavras-chave na tabela de símbolos é apenas uma entre várias que podem ser utilizadas. Dependendo de qual deve ser o trabalho a ser realizado, o próximo capítulo estudará estruturas de dados mais sofisticadas para tabelas de símbolos. Em projetos de compiladores de programação e de outros domínios, enquanto que, usando-se esta ou aquela, pode-se ter que uma cadeia de estados produza o efeito desejado, sempre há uma delas que é a mais natural e eficiente.

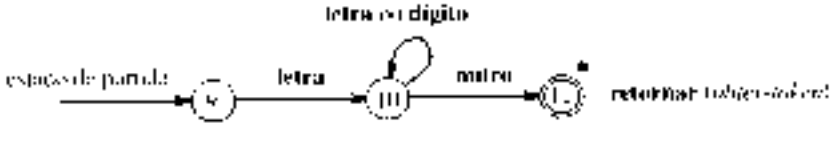


Fig. 3.13. Diagrama de transições para identificadores e palavras-chave

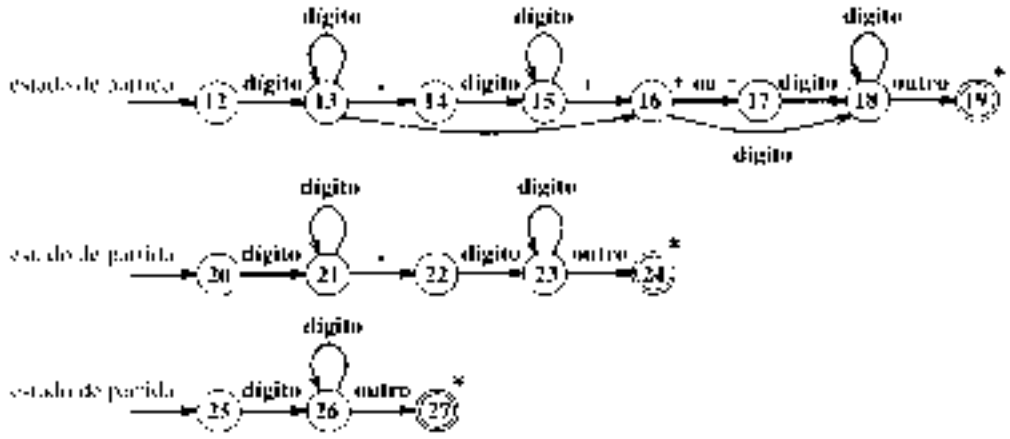


Fig. 3.14. Diagramas de transições para tokens semelhantes aos da linguagem Pascal

Exemplo 3.9. Vamos tentar entregar a você a tarefa de escrever um reconhecedor para números sem sinal, dados pela seguinte definição regular:

$$\mathbf{num} \rightarrow \mathbf{dígitos}\ (.\ \mathbf{dígitos})?\ (\mathbf{E}\ (+\mid -)?\ \mathbf{dígitos})?$$

Note que a definição é da forma **dígitos fração? expoente?**, na qual **fração** e **expoente** são opcionais.

O leitor pode estar se perguntando se existe uma forma mais fácil de escrever um reconhecedor para números sem sinal. Por exemplo, por que não podemos simplesmente esperar até encontrar um caractere que não seja um dígito, um ponto ou a letra E, e então chamar uma função como a *retrocede* para devolver o caractere lido à entrada? Começando pelos estados 25, 20 e 12 na Fig. 3.14, os estados de aceitação serão atingidos depois de um, dois e três dígitos, respectivamente, exceto quando for seguido por um outro dígito. Os diagramas de transição com os estados de partida 25, 20 e 12 são para **dígitos**, **dígitos fração** e **dígitos fração? expoente**, respectivamente. Consequentemente, os estados de partida precisam ser levados ao algoritmo de simulação 12, 20 e 25.

A ação quando qualquer um dos estados de aceitação 19, 24 ou 27 for atingido é chamar um procedimento *retrocede* (como aquele do exemplo de leitura de buffer) que devolve o apontador para a entrada, *avante*. O analisador léxico retorna o valor gerado atualmente associado a **num** para o corpo do valor léxico. ■

As informações sobre a linguagem especificada pelas definições regulares dos tokens podem ser usadas para assinalar erros de entrada. Por exemplo, com a entrada 1.E, falharemos nos estados 14 e 22 da Fig. 3.14, como o próximo caractere de entrada não for um dígito ou um dos números esperados quiserem reconhecer um erro e continuarem como se a entrada fosse 1.0E. Tal complicação pode também ser usada para simplificar diagramas de transições, porque o tratamento de erros pode ser usado para se recuperar de algumas situações que, de outra forma, levariam a uma falha.

Existem várias formas pelas quais os erros podem ser tratados. Uma delas é garantir que nenhum dos estados de aceitação em um diagrama de transições, como o da Fig. 3.14, possa ser atingido. Um erro que é fácil de cometer ao escrever os diagramas é esquecer um estado de aceitação. Outro é o de mudar a especificação no caso de uma falha no processo de se acompanhar o diagrama. Uma abordagem explorada mais tarde neste capítulo nos permite gerar, a partir de várias especificações de alto nível, diagramas de transições que não contêm estados de aceitação inacessíveis.

Exemplo 3.10. Uma sequência de diagramas de transições para todos os tokens do Exemplo 3.6 é obtida se juntarmos os diagramas das Figs. 3.12, 3.13 e 3.14. Estados de partida com numeração mais baixa devem ser colocados antes nessa lista do que estados com numeração mais alta.

Os únicos lexemas remanescentes são relacionados aos espaços em branco. O diagrama de transições para o espaço em branco é diferente daqueles dos padrões descritos acima, porque nada é retornado ao parser quando o espaço em branco é encontrado na entrada. Um diagrama de transições para reconhecer um ou mais espaços em branco é:

Nada é retornado quando o estado de aceitação é atingido; novamente retornamos para o estado de partida do primeiro diagrama de transições a fim de procurar por outros padrões.

Sempre que possível, é melhor procurar por tokens que ocorrem frequentemente antes daqueles que ocorrem menos frequentemente, porque um diagrama de transições é atingido somente após serem falhados os diagramas anteriores. Como o espaço em branco é esperado ocorrer frequentemente, a colocação do diagrama de transições para espaço em branco próximo do início deve demonstrar ser um melhoramento em relação à construção atual. ■

Implementando um Diagrama de Transições

Uma sequência de diagramas de transições pode ser convertida num programa, que procura pelos tokens especificados pelos diagramas. Adotamos uma estratégia sistemática que funciona para todos os diagramas de transições e constrói programas cujo tamanho seja proporcional ao número de estados e lados dos diagramas.

Cada estado recebe um segmento de código. Se existem lados deixando um estado, então seu código lê um caractere e seleciona um lado para seguir, se possível. A função *proximo_caractere* é usada para ler o próximo caractere a partir do *buffer* de entrada, avançando o apontador *avante* a cada chamada e retornando o caractere lido. Se existe um lado rotulado pelo caractere lido ou por uma classe de caracteres que o contenha, o controle é, então, transferido para o código do estado apontado por aquele lado. Se não existe tal lado, o estado corrente não é um daqueles que indica que um token foi encontrado e, nesse caso, a rotina *falha* é invocada para retornar o apontador *avante* para a posição de

Fig. 3.15 Código C para a declaração e inicialização de uma variável global

único e conveniente é a busca por meio de uma tabela de palavras-chave. Nessa tabela, cada palavra-chave é armazenada com um código que identifica o token correspondente. Quando o analisador léxico encontra um identificador, ele primeiro verifica se o identificador é uma palavra-chave, procurando-o na tabela de palavras-chave. Se o identificador for encontrado na tabela, o código do token correspondente é retornado. Caso contrário, o identificador é tratado como um identificador definido pelo usuário.

Para ilustrar essa técnica, vamos supor que a palavra-chave `int` tenha o código 258, a palavra-chave `float` tenha o código 259 e a palavra-chave `if` tenha o código 260. Quando o analisador léxico encontra a palavra `int`, ele retorna o código 258. Quando encontra a palavra `float`, ele retorna o código 259. Quando encontra a palavra `if`, ele retorna o código 260. Quando encontra qualquer outra palavra, ele retorna o código 261, que é o código para identificadores definidos pelo usuário.

Essa técnica é muito eficiente e fácil de implementar. Ela também é flexível, pois permite que novas palavras-chave sejam adicionadas à tabela sem a necessidade de modificar o código do analisador léxico.

Aqui está um exemplo de como essa técnica pode ser implementada em C:

```
#include <stdio.h>
#include <string.h>

#define NUM_KEYWORDS 3

struct keyword {
    char *word;
    int token;
};

struct keyword keywords[NUM_KEYWORDS] = {
    {"int", 258},
    {"float", 259},
    {"if", 260}
};

int lookup(char *word) {
    int i;
    for (i = 0; i < NUM_KEYWORDS; i++) {
        if (strcmp(word, keywords[i].word) == 0) {
            return keywords[i].token;
        }
    }
    return 261; /* identificador definido pelo usuário */
}

int main() {
    char word[100];
    while (scanf("%s", word) != EOF) {
        int token = lookup(word);
        printf("token: %d\n", token);
    }
    return 0;
}
```

Esse código define uma estrutura `keyword` que contém uma palavra-chave e seu código de token correspondente. A função `lookup` procura uma palavra na tabela de palavras-chave e retorna o código de token correspondente, se a palavra for encontrada. Caso contrário, ela retorna o código 261, que é o código para identificadores definidos pelo usuário.

A função `main` lê palavras da entrada padrão e imprime o código de token correspondente para cada palavra. Se a palavra for uma palavra-chave, o código de token correspondente será impresso. Caso contrário, o código 261 será impresso.

Essa técnica é muito eficiente e fácil de implementar. Ela também é flexível, pois permite que novas palavras-chave sejam adicionadas à tabela sem a necessidade de modificar o código do analisador léxico.

3.5 UMA LINGUAGEM PARA ESPECIFICAÇÃO DE ANALISADORES LÉXICOS

Várias ferramentas têm sido construídas para os analisadores léxicos a partir de descrições de padrões especificados usando-se expressões regulares. Já vimos o uso das expressões regulares para especificar os padrões

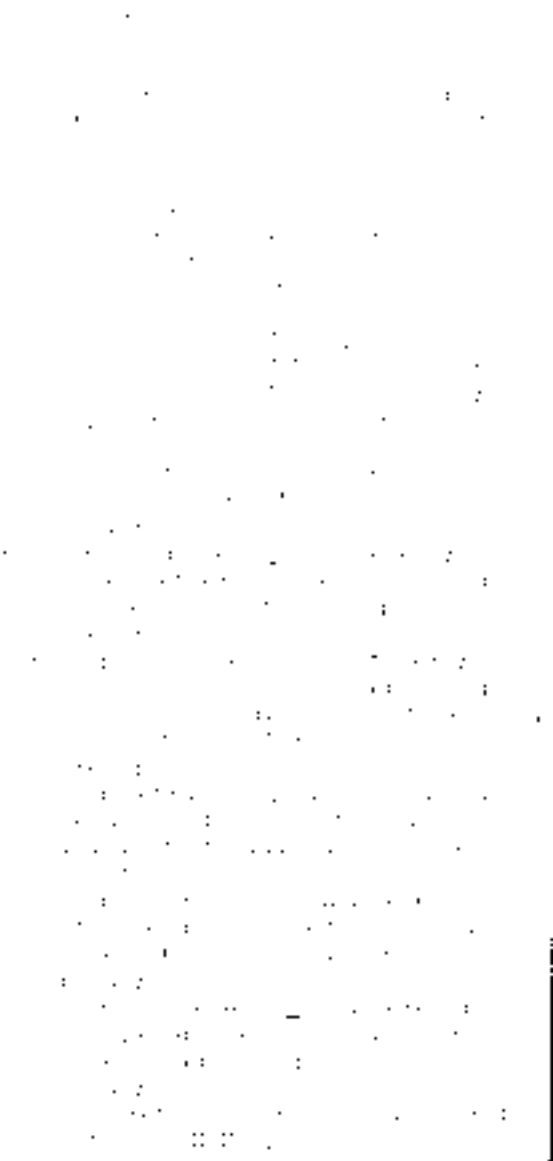


Fig. 3.16 Código C para a função `lookup`

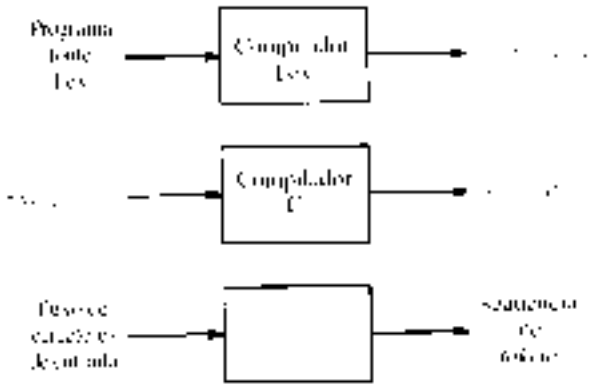


Fig. 3.17. Criando um tradutor com Lex e Yacc

O programa consiste em uma representação tabular de um diagrama de transições construído a partir das expressões regulares do . Juntamente com a tabela é colocada uma rotina padrão que trata a tabela e usa-a para reconhecer os lexemas. As ações associadas a cada expressão regular são fragmentos de código escritos em que serão executados tão logo o lexema seja reconhecido. Tipicamente, o código associado permite que o reconhecedor léxico retorne o controle ao seu chamador, após colocar o nome do token e outros valores associados. Se for encontrada uma cadeia de entrada não associada a nenhum , o analisador léxico gerado por entra em uma ação default, que normalmente transfere um caractere da entrada para a saída e procura pelo próximo lexema. Em geral, o código é escrito de forma a economizar caracteres lidos além do próximo lexema.

Especificações Lex

Um programa Lex é constituído de três partes:

```
declarações
%%
regras de tradução
%%
procedimentos auxiliares
```

A seção de declarações inclui declarações de variáveis, de constantes manifestas e definições regulares. (Uma constante manifesta é um identificador declarado para representar uma constante.) As definições regulares são usadas como componentes das expressões regulares que aparecem nas regras de tradução.

As regras de tradução de um programa Lex são enunciados da forma

```
p1  {ação 1}
p2  {ação 2}
...
pn  {ação n}
```

onde cada é uma expressão regular e cada é uma fragmento de programa descrevendo que ação o analisador léxico gerado deverá tomar quando padrão for reconhecido em um lexema. Em Lex as ações são escritas em C, em geral, porém, podem ser em qualquer outra linguagem de implementação.

Uma terceira seção contém quaisquer procedimentos auxiliares que sejam necessários pelas ações. Alternativamente, esses procedimentos podem ser compilados separadamente e carregados com o analisador léxico.

Um analisador léxico criado por Lex se comporta em associação com um parser da seguinte forma. Quando ativado pelo parser, o analisador léxico começa lendo sua entrada remanescente, um caractere de cada vez, até que tenha encontrado o mais longo prefixo da entrada que seja reconhecido por uma das expressões regulares . Em seguida executa a ação . Tipicamente, a ação retornará o controle para o parser. Entretanto, se assim não for, o analisador léxico prosseguirá até encontrar mais lexemas que levem a uma ação que retorne o controle ao parser. A pilha sintática do parser, pois lexemas até um ponto de retorno explícito ou implícito. Em seguida, o analisador léxico coloca o restante do lexema em um buffer de entrada, onde começará o próximo reconhecimento.

O analisador léxico também armazena quatro valores em variáveis externas. A função pode passar um valor de atributo com informações sobre o lexema, podendo também colocar valores em variáveis globais definidas

Exemplo 3.14. A Fig. 3.18 é um programa Lex que reconhece os tokens da Fig. 3.12 e retorna a função encontrado. Várias pontas observações respeito do código ajudam a entender algumas características importantes de Lex.

Na seção de declarações, vemos uma região para a declaração de certas constantes manifestas usadas pelas regras de tradução. Essas declarações são envolvidas por caracteres e pessoas [1]. Qualquer coisa digna, mas entre essas chaves é copiada diretamente no programa léxico . Também, qualquer texto tratado como parte das definições regulares ou das regras de tradução, exatamente o mesmo tratamento é dado para os procedimentos auxiliares na terceira seção. Na Fig. 3.18, existe um único procedimento, , que instala o lexema encontrado na tabela de símbolos.

Também incluídas na seção de definições estão algumas definições regulares. Cada tal definição consiste em um nome e uma expressão regular denotada pelo nome. Para evitar confusão entre o primeiro nome, o segundo e a figura de linguagem, cada um de caracteres , , ou qualquer dos três símbolos, espaço, tabulação ou representado por . A segunda definição é a de espaço em branco, denotada pelo nome . Espaço em branco é qualquer sequência de um ou mais caracteres de branco. Note que a palavra não precisa ser envolvida por chaves em Lex. A falta de distinção do padrão é constituído pelos meta-lexemas .

Na definição de , o corpo é uma classe de caracteres. A forma simplificada significa qualquer uma das letras maiúsculas, de a . Assim, as letras podem ser qualquer sequência de uma ou mais caracteres de letras maiúsculas. Note que a palavra não precisa ser envolvida por chaves em Lex. A falta de distinção do padrão é constituído pelos meta-lexemas .

Na última definição regular, a de , vemos alguns pontos detalhes a mais. Vemos usado com um meta-símbolo com seu significado usual de "zero ou mais ocorrências de". Também vemos a barra invertida usada como um caractere de escape. Um de determinado caractere que é um meta-símbolo Lex tem seu significado natural. Em particular, o ponto decimal na definição de é expresso por porque um ponto por si só é reconhecido a classe de todos os caracteres exceto a mudança de linha. Tanto em Lex quanto em muitos outros programas do sistema UNIX que idiomas com expressões regulares. Na classe de caracteres , colocamos uma barra invertida antes do sinal de menos porque o mesmo figurando isolado poderia ser confundido com seu uso para denotar um intervalo, como em .

Existe uma outra forma de se fazer com que os lexemas tenham seus significados próprios, ainda que sejam meta-símbolos de Lex: cercá-los entre aspas. Mostramos um exemplo dessa convenção na seção de regras de tradução, onde seus operadores relacionais foram cercados por aspas.

---

[1] Na verdade, o programa Lex simplesmente copia tudo entre os dois primeiros %% que encontra, mas não se deve contar com isso. Se você quiser que suas declarações sejam copiadas, coloque-as entre e .

[2] Em Lex, a classe de caracteres [^] é usada para denotar todos os caracteres exceto um dado. Se o único caractere for , então a classe é todos os caracteres exceto . Lex também tem uma classe de caracteres [ ] que denota todos os caracteres exceto e . Novamente, necessidade de colocar a barra invertida antes do sinal de menos.

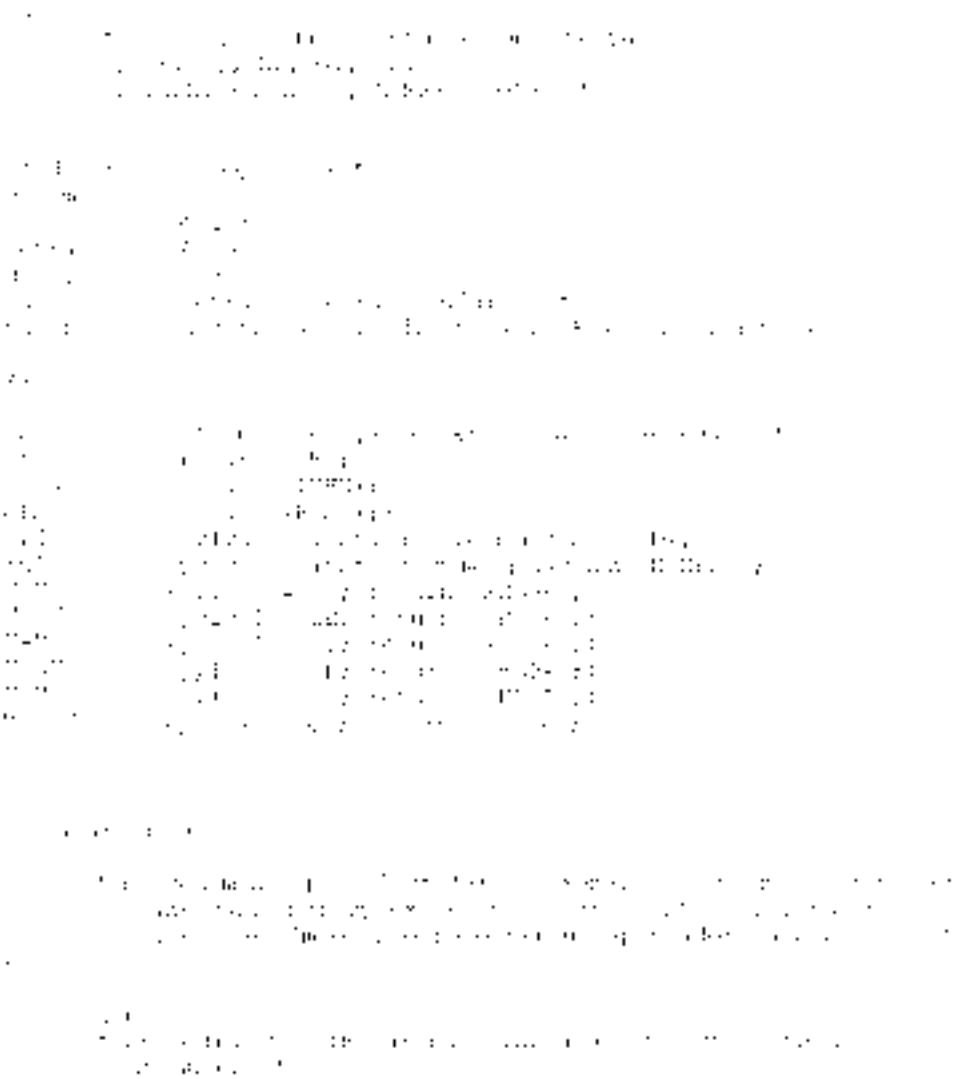


Fig. 3.18 Programa Lex para o Exemplo da Fig. 3.16.

Agora, vamos considerar as regras de tradução na seção que se segue ao primeiro `%%`. A primeira regra diz que se encontrarmos um espaço, tabulação ou caractere de nova linha, não faremos qualquer ação. Em particular, não retornamos ao parser. Relembremos que a execução do analisador léxico só é feita de tal forma que o mesmo se mantém chamando reconhecer *tokens* até que uma ação associada com um token encontrado passe um retorno.

A segunda regra diz que se as letras `if` forem vistas, deve-se retornar o token `IF` que é uma constante manifesta representando algum inteiro compreendido pelo parser como sendo o *token* **if**. As duas regras seguintes tratam das palavras-chave `then` e `else`, similarmente.

Na regra para **id**, vemos dois importantes detalhes. Primeiro, a variável `yylval`, que é estabelecida como o valor retornado pelo procedimento `yylex()`, é definida como sendo o procedimento instalID(). Segundo, `yytext` é uma variável cuja definição aparece na saída do Lex, `lex.yy.c`, e que fica em nosso dispositivo para o parser. O propósito de instalID() é de gravar o valor léxico reconhecido, uma vez que a seguinte instrução `return(ID)` pode somente retornar um código para a classe do token.

Não mostraremos os detalhes do código de instalID(). Internamente, podemos supor que o mesmo procure na tabela de símbolos pela sequência de símbolos dada pelo padrão **id**. Uma entrada é encontrada disponível para a mesma, se ela ainda não estiver lá, através das duas variáveis `yytext` e `yyleng`. A variável `yylval` corresponde à variável que temos chamado de *tokenval*, de modo que seja um apontador para a posição da tabela de símbolos. Por exemplo, se `yytext` falhar em encontrar o identificador na tabela de símbolos, poderá criar uma nova entrada para o mesmo. Depois, `yylval` poderá ser apontado para essa entrada. Como antes, poderemos ser requisitados por outros componentes a delimitar por um marcador de fim de cadeia, como na Seção 2.7. A nova entrada na tabela de símbolos apontada para o mesmo é essa cópia.

Os números são tratados similarmente pela regra seguinte, e para as últimas duas regras, `yylval` é usada para informar ao gerador de código para o operador relacional particular encontrado, enquanto o valor efetivo de retorno é o código para o token **relop** em cada caso.

Suponhamos que um dado usuário Lex cria resultante do programa da Fig. 3.18 seja fornecida uma entrada consistindo de duas tabulações, os letras `if` e um espaço. As duas tabulações são o mais longo prefixo inicial da entrada reconhecido por um padrão, especificamente entre o padrão

A ação para `ws` é de não realizar coisa alguma e, então, o analisador lexical inicia a operação de leitura de letras, para `if`, e começa a procurar por outros *tokens*.

O próximo lexema a ser reconhecido é `then`. Suponhamos que os padrões `id`, `if` e `then` reconheçam esse lexema e nenhum dos dois seguintes reconhece uma cadeia mais longa. Como o padrão para a palavra-chave `if` precede o padrão para os identificadores na lista da Fig. 3.18, o conflito

...é a cadeia de caracteres da palavra-chave. Em geral, esta estratégia de reconhecimento de cadeias é muito útil sempre que palavras-chave forem usadas como padrões para os identificadores.

Como um outro exemplo, suponhamos que = seguido por dois quaisquer caracteres brancos seja um padrão para o reconhecimento de um certo comentário, e que um padrão mais longo que reconhece um comentário mais longo também exista. Consequentemente, a estratégia de Lex é selecionar o padrão mais longo reconhecido por um padrão, facilitando a resolução de conflitos entre = e = ... = da forma esperada. Por exemplo, a seleção de = como o padrão mais longo...

O Operador *Lookahead*

...dos na Seção 3.4, os analisadores léxicos podem ser construídos para muitas linguagens de programação. Precisamos examinar além do fim de lexemas antes que um padrão possa ser determinado. Lembre-se do exemplo da Fortran, apresentado no par de comandos

...

Com espaços em branco insignificantes, o lexema para IF pode ser qualquer prefixo de IF(I,J,K)=..., que leva a um dos dois comandos. Ou seja, não podemos dizer até que tenhamos examinado o caractere após o) se IF é uma palavra-chave ou o nome de um vetor. Se assumirmos que IF é uma palavra-chave, ela é reconhecida quando o padrão...

...

No entanto, quando o analisador léxico é construído a partir de um diagrama de transições, é muito difícil detectar o fim de uma palavra-chave, pois geralmente existem muitas palavras-chave que podem ser prefixos de outras palavras-chave ou de identificadores.

Em Lex, podemos escrever um padrão para cada palavra-chave, usando expressões regulares semelhantes às do Exemplo 3.10. Supondo que essas palavras-chave sejam todas reconhecidas como tokens, que também são usados para reconhecer identificadores, a ambiguidade sobre palavras-chave e identificadores pode ser resolvida colocando-se os padrões para palavras-chave antes do padrão para identificadores. Infelizmente, isso significa que palavras-chave reservadas, como IF no exemplo anterior, são reconhecidas como palavras-chave mesmo quando não são usadas como palavras-chave.

Com essa especificação, o analisador léxico do Lex irá prosseguir até ter certeza de que viu um parêntese depois de uma cadeia de letras e dígitos, seguida por um sinal de igual, seguida por outra cadeia de letras e dígitos, seguida por uma vírgula. Não importa quão longe tenha de procurar, ou quantas linhas de entrada tenham de ser examinadas. Em particular, ele pode ter de passar muitos tokens IDENTIFIER antes de encontrar a vírgula que revela que IF não era uma palavra-chave, afinal. Em tais situações, o analisador léxico também precisará armazenar a entrada lida até que uma decisão possa ser tomada.

Exemplo 3.12. O operador *lookahead* pode ser usado para solucionar um outro difícil problema de análise léxica em Fortran: distinguir palavras-chave de identificadores. Por exemplo, a entrada

...

pode ser uma atribuição perfeitamente válida em Fortran, e não um comando lógico IF. Um token de especificação para a cadeia de caracteres DO pode ser encontrado mais adiante na entrada, mesmo em programas razoáveis. Por exemplo, a entrada

...

Escrevemos 73 não deixando uma outra forma de erro tão ruim. Um erro...

...

Notamos que cada comando DO termina em um rótulo seguido por uma letra e que cada parêntese à direita usado para a subscrição ou a agrupamento de operandos precisa ser seguido por um símbolo de operação, tal como +, em nosso exemplo. Logo, o parêntese à direita após 1.25 não pode ser o último caractere de um comando DO válido. Um parêntese à direita após uma letra pode ser seguido por uma letra. Nesta situação, a única coisa que pode ser...

...

O ponto fixo na linguagem de "qualquer caractere menos vírgula" e os barra-jota são tratados à frente dos parênteses. Lex é um gerador de analisadores léxicos para traduzir essas cláusulas diretamente em autômatos finitos determinísticos, que reconhecem expressões regulares (ver Exercício 3.10).

Outra forma de atacar o problema imposto pelos comandos IF em Fortran e outras construções é deixar o analisador léxico declarar todas as palavras-chave como tokens. Especificamente, o padrão completo é dado a ele se nenhum outro padrão for satisfeito. Tais testes tornam mais difícil a implementação eficiente de um analisador léxico a partir de uma especificação. Lex e padrões mais simples para especificar analisadores léxicos são úteis para muitas aplicações. Frequentemente, é necessário fazer um pouco mais, como a simulação de diagramas de transições, a fim de determinar se algum deles é necessário. Deve ser notado que a linguagem Fortran não pode ser totalmente analisada com a tecnologia regular que descrevemos neste capítulo. Em vez disso, um analisador léxico é feito para uma linguagem convencional de programação, como C ou Java, que usa um pequeno autômato de análise léxica.

3.6 AUTÔMATOS FINITOS

Um reconhecedor para uma linguagem é um programa que recebe como entrada uma cadeia *x* e responde "sim" se *x* é uma sentença da linguagem e "não" em caso contrário. Compilamos expressões regulares dentro de reconhecedores construindo, de um diagrama de transições generalizado chamado de autômato finito. Um autômato finito pode ser determinístico ou não-determinístico, onde "não-determinístico" significa que mais de uma transição para fora de um estado pode ser possível para o mesmo símbolo de entrada.

Tanto os autômatos finitos determinísticos quanto os não determinísticos são capazes de reconhecer precisamente os conjuntos regulares. Convenientemente, ambos podem reconhecer exatamente o que as expressões regulares podem definir. Entretanto, existe uma troca de tempo-espaço: autômatos finitos determinísticos podem levar a reconhecedores muito mais rápidos do que os não determinísticos, mas um autômato finito determinístico pode ser muito maior do que um autômato finito não determinístico equivalente. Na próxima seção, apresentamos métodos para converter expressões regulares em ambos os tipos de autômatos finitos. A conversão para um autômato finito não determinístico é mais direta e também dá origem a um dos algoritmos mais importantes para a compilação de padrões.

Os exemplos desta seção e da próxima não pertencem diretamente a uma linguagem de entrada pela expressão regular. Por vezes, consistem em conjuntos de todas as cadeias de 0's e 1's terminadas em 101. Tais cadeias surgem na prática. Por exemplo, um processo

regular para os tokens de forma mais simples e que reconhece a forma da forma [...] como uma representação para um caractere que não um ponto ou nova linha. Existem também os símbolos especiais ^ e $ que casam com o início e o fim de uma linha, respectivamente. Para mais detalhes, consulte a página de manual do grep.

Autômatos Finitos Não-Determinísticos

Um *autômato finito não-determinístico* (AFN) simplificadamente pode ser visto como um modelo matemático que consiste em:

1. um conjunto de *estados* S;
2. um conjunto de símbolos de entrada Σ, os *símbolos de transição* válidos;
3. uma função de transição que, *recebendo*, que mapeia pares estado-símbolo em conjuntos de estados;
4. um estado s_0 que é distinguido como o *estado de partida* (ou *estado inicial*);
5. um conjunto de estados F distinguidos como *estados de aceitação*, ou *finais*.

Um AFN pode ser representado graficamente por um grafo dirigido e rotulado, chamado de *grafo de transições*, no qual os nós são os estados e os lados orientados representam a função de transição. Esse grafo se parece com um diagrama de transições, mas o mesmo rótulo pode aparecer em duas ou mais transições para fora do mesmo estado, e os lados podem ser rotulados pelo símbolo especial ε bem como pelos símbolos de entrada.

Um grafo de transições para um AFN que reconhece a linguagem gerada por (*a*|*b*)*a*b*b é mostrado na Fig. 3.19. O conjunto de estados do AFN é $\{0, 1, 2, 3\}$ e o alfabeto de símbolos de entrada é $\{a, b\}$. Os estados 0 e 3 da Fig. 3.19 são distinguidos como estado de partida e estado de aceitação, respectivamente, como indicado por um círculo duplo.

Ao descrever um AFN, usamos a representação de grafos de transições. Como vimos, a função de transição de um AFN pode ser implementada de várias formas diferentes pelo computador. A implementação mais fácil é a *tabela de transições*, na qual existe uma linha para cada estado e uma coluna para cada símbolo de entrada e para ε, respectivamente. A entrada para a linha de símbolo a e coluna labels é o conjunto de estados com mais provavelmente na prática, um apontador para um conjunto de estados que pode ser alcançados através de uma transição a partir do estado e entrada a. A tabela de transições para o AFN da Fig. 3.19 é mostrada na Fig. 3.20.

ESTADO	SÍMBOLO DE ENTRADA	
	a	b
0	{0, 1}	0
1	—	{2}
2	—	{3}

Fig. 3.20. Tabela de transições para o autômato finito da Fig. 3.19.

A representação sob a forma de tabela de transições possui a vantagem de possibilitar um acesso rápido às transições de um dado estado e caractere. Sua desvantagem é que pode ocupar um grande espaço quando o alfabeto de entrada for grande e a maioria das transições for para o conjunto vazio. Representações da função de transição sob forma de listas de arcos de transição podem ser mais econômicas em termos de espaço, mas consomem mais tempo para localizar as transições. Vamos ficar sabendo que podemos facilmente converter uma qualquer uma dessas implementações de um autômato finito para outra.

Um AFN aceita uma cadeia de entrada x se e somente se existir algum percurso no grafo de transições a partir do estado inicial até algum estado de aceitação, tal que os rótulos dos lados ao longo do percurso soletrem a cadeia x. O AFN da Fig. 3.19 aceita as cadeias de entrada *aabb*, *ababb*, *aaabb*, ... Por exemplo, *aabb* é aceita pelo percurso pontilhado, que passa pelo estado rotulado 0 duas vezes, depois pelo estado 1, e então, para os estados 2 e 3 através dos lados rotulados com *b*, respectivamente.

Um percurso pode ser representado por uma sequência de pares de estado e símbolo chamados *movimentos*. A sequência de movimentos realizados ao se aceitar a cadeia de entrada *aabb* é

$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

Em geral, uma cadeia de uma sequência de movimentos pode levar a muitos estados de aceitação. Nesse caso, diz-se que a cadeia é aceita se existir pelo menos uma dessas sequências que leve a um estado de aceitação. Por exemplo, uma outra sequência válida de movimentos para a cadeia *aabb* é

$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$

A *linguagem definida* por um AFN é o conjunto de todas as cadeias de entrada que o mesmo aceita. Não é difícil mostrar que o AFN da Fig. 3.19 aceita a *linguagem* (*a*|*b*)**abb*.

Exemplo 3.13. Na Fig. 3.21 vemos um AFN para reconhecer a linguagem (*a*|*b*)**abb*. A cadeia *aabb* é aceita através da sequência dos estados 0, 1, 2, 3. Os rótulos desses lados são *a*, *a*, *b*, *b*, respectivamente. Note que os lados "desaparecem" na concatenação.

Autômatos Finitos Determinísticos

Um *autômato finito determinístico* (AFD) é um caso especial de um AFN que não possui transições-ε e no qual

1. nenhum estado possui uma transição-ε, isto é, uma transição ε em todas as entradas, e
2. para cada estado s e símbolo de entrada a, existe no máximo um lado rotulado a deixando s.

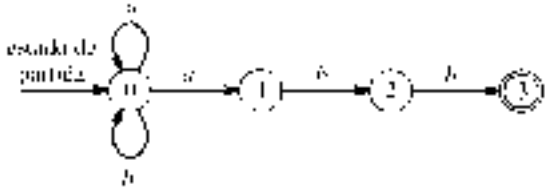


Fig. 3.19. Um autômato finito não-determinístico.

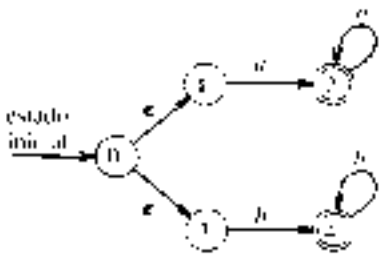


Fig. 3.21. AFN que aceita (*a*|*b*)**abb*.

Um autômato finito possui exatamente uma transição, a partir de cada estado, para qualquer símbolo de entrada. Se este for o caso, é uma tarefa direta construir uma tabela de transições para representar a função de transição de um AFD: cada célula da tabela de transições será um único estado. Uma consequência é que é fácil determinar se um autômato finito determinístico aceita uma cadeia de entrada, já que existe no máximo um caminho rotulado especificando a cadeia a partir do estado inicial. O algoritmo seguinte mostra como simular o comportamento de um AFD, dada uma cadeia de entrada.

Algoritmo 3.1. Simulando um AFD.

Entrada. Uma cadeia x terminada por um caractere de fim de arquivo **eof**. Um AFD D com estado de partida s_0, conjunto de estados de aceitação F e função de transição $move$.

Saída. A resposta "sim" se D aceita x; "não" em caso contrário.

Método. Aplique o algoritmo da Fig. 3.22 para a cadeia de entrada x. A função $estado\_vazio(s, a)$ fornece o estado para o qual existe uma transição a partir do estado s com a entrada a. A função $proximo\_caractere()$ lê um próximo caractere da cadeia de entrada x.

Exemplo 3.14. Na Fig. 3.23 vemos um grafo de transições de um autômato finito determinístico que aceita a mesma linguagem reconhecida pelo AFN da Fig. 3.19. Com este AFD e a cadeia de entrada $aabb$, o Algoritmo 3.1 entra as sequências de estados $0, 1, 2, 3$ e 4, e então pára.

```
s = s0
c = proximo_caractere()
enquanto c ≠ eof faça
        s = estado_vazio(s, c)
        c = proximo_caractere()
fim-enquanto
se s está em F então
        retornar "sim"
senão retornar "não"
```

Fig. 3.22. Simulação de um AFD.

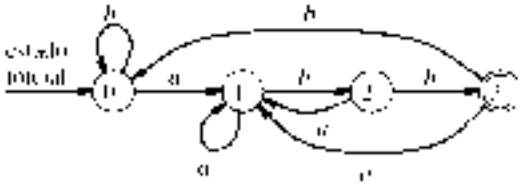


Fig. 3.23. AFD que aceita $(a|b)^*abb$.

Conversão de um AFN num AFD

Note se que o AFN da Fig. 3.19 possui duas transições do estado 0 com a entrada a: seja para o estado 0 ou o estado 1. Sim lamente o AFN da Fig. 3.21 possui duas transições com a a partir do estado 0. Conquanto o fato não seja mostrado no exemplo, a mesma situação onde podemos escolher entre uma transição com a ou outra com o símbolo real de entrada também ocorre ocasionalmente. Nessas situações, nenhuma função de transição pode ser calculada, tornando difícil simular um AFN com um programa de computador. A definição de aceitação meramente estabelece que pelo menos um caminho rotulado pela cadeia de entrada em questão vai de um estado inicial até um estado de aceitação. Mas se existem vários caminhos que satisfazem a mesma cadeia de entrada, pode ser difícil saber durante a simulação quais deles devem ser escolhidos, ou mesmo se é preciso explorar mais de um deles.

Apresentamos agora um algoritmo para construir um AFD a partir de um AFN que reconhece a mesma linguagem. Esse algoritmo, frequentemente chamado de *construção de subconjuntos*, é útil na simulação de um AFN por um programa de computador. Um algoritmo relacionado pode ser usado para minimizar o número de estados de um AFD (veja o Exercício 3.9.8).

Na tabela de transições de um AFN, cada entrada é um conjunto de estados; na tabela de transições de um AFD, cada entrada é exatamente um único estado. A ideia geral por trás da construção AFN-para-AFD é que cada estado do AFD corresponde a um conjunto de estados do AFN. O AFD usa seu estado para controlar todos os possíveis estados que o AFN poderia estar após ler cada símbolo de entrada. Intuitivamente, após ler a entrada $a_1 a_2 \cdots a_n$, o AFD estará num estado que representa o subconjunto S dos estados do AFN que são alcançáveis a partir do estado inicial do AFN inteiramente com algum percurso rotulado $a_1 a_2 \cdots a_n$. O número de estados do AFD pode ser até exponencial do número de estados do AFN, mas na prática esse caso extremo ocorre raramente.

Algoritmo 3.2. *Construção de subconjuntos.* Construindo um AFD a partir de um AFN.

Entrada. Um AFN N.

Saída. Um AFD D que aceita a mesma linguagem.

Método. Nosso algoritmo constrói uma tabela de transições $Dtran$ para D. Cada estado do AFD é um conjunto de estados do AFN e construímos $Dtran$ de tal forma que D se ponha em "paralelo" todos os possíveis movimentos que N possa fazer a uma dada cadeia de entrada.

Usamos as operações da Fig. 3.24 para controlar os conjuntos de estados do AFN que representam um estado do AFD e T o conjunto de estados do AFN.

Operação	Descrição
fecho-vazio(s)	Conjunto de estados do AFN alcançáveis a partir do estado s do AFN somente através de transições-ε
fecho-vazio(T)	Conjunto de estados do AFN alcançáveis a partir de algum estado s do AFN pertencente a T somente com transições-ε
mover(T, a)	Conjunto de estados do AFN para o qual existe uma transição com símbolo de entrada a a partir de algum estado s do AFN, pertencente a T

Fig. 3.24. Operações sobre os estados do AFN.

Operações $fecho\text{-}vazio(t)$ no conjunto de estados de D, e $Dtran$, a tabela de transições para D, da seguinte maneira. Cada estado de D corresponde a um conjunto de estados do AFN que podem ser atingi-

```
inicialize Dtran[T, a] como o conjunto vazio e todos os estados de D estão
de estado não marcado; enquanto existir um estado T não
marcado em Estados-D faça início
    marque T;
    para cada símbolo de entrada a faça início
        U := fecho-transições(mover(T, a));
        se U não está em Estados-D então
            adicione U como um estado não marcado de
Estados-D
        Dtran[T, a] := U
    fim
fim
```

Fig. 3.25. Construção de subconjuntos.

gados em N após ter algumas sequência de símbolos de entrada, em particular, todos os estados acessíveis por transições vazias antes ou depois dos símbolos terem sido lidos. O estado de partida de D é *fecho-transições*(s_0), e os estados e transições são adicionados a D usando o algoritmo da Fig. 3.25. Um estado de D é de aceitação se for um conjunto de estados do AFN contendo pelo menos um estado de aceitação de N.

O computo de *fecho-transições*(T) é um processo típico de busca dos nós acessíveis a partir de um dado conjunto de nós num grafo. Neste caso, os estados de T são o conjunto de nós iniciais, e seguimos exatamente os lados rotulados ε no AFN. Um algoritmo simples para computar *fecho-transições*(T) usa uma pilha para guardar os estados que ainda não foram checados pela existência de lados rotulados ε saindo deles. Tal processamento é mostrado na Fig. 3.26.

Exemplo 3.15. A Fig. 3.27 mostra o AFN N que reconhece a linguagem gerada pela expressão regular `(a|b)*abb`. Acontece que o mesmo AFN pode ser construído regularmente a partir da expressão regular. Vamos aplicar o Algoritmo 3.2 a N. O estado inicial de AFD equivalente é *fecho-transições*(0), que é $A = \{0, 1, 2, 4, 7\}$, porque esses são exatamente os estados acessíveis a partir do estado 0 através somente de transições ε. Note que a ordem em que os estados aparecem em A não é importante. Também é importante notar que todo conjunto de estados que contém o estado 8 é o estado de aceitação de N.

O alfabeto de símbolos de entrada aqui é $\{a, b\}$. O algoritmo da Fig. 3.25 nos diz para marcar A e, em seguida, computar

$$b_1 = \textit{fecho-transições}(\textit{mover}(A, a))$$

Para computar *mover*(A, a), o conjunto de estados de A com transições em a para partes iniciais do estado 3. Dentre os estados 0, 1, 2, 4 e 7, somente 2 e 7 têm tais transições, para 3 e 8, e assim

$$b_1 = \textit{fecho-transições}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\}$$

Vamos chamar esse conjunto de B. Então $Dtran[A, a] = B$.

```
coloque todos os estados em T na pilha;
inicialize fecho-transições com T (cada estado é
acessível a partir de si mesmo);
enquanto pilha não estiver vazia executar início
    desempilhar t, o topo da pilha;
    para cada estado u com um lado de t para o estado u
    executar se u não estiver em fecho-transições faça início
        adicione u a fecho-transições;
        empilhar u na pilha
    fim
fim
```

Fig. 3.26. Computo de *fecho-transições*.

Dos estados em A, somente 4 possui uma transição em b para 5, e consequentemente $Dtran[A, b] = \{1, 2, 4, 5, 6, 7\}$.

$$C = \textit{fecho-transições}(\{5\}) = \{1, 2, 4, 5, 6, 7\}$$

Dessa forma, $Dtran[A, b] = C$.

Se continuarmos esse processo com os conjuntos ainda não marcados B e C, acabaremos eventualmente o ponto onde todos os conjuntos que se pode obter estão marcados. Isto é garantido porque existem "somente" 2^{11} subconjuntos de um conjunto de onze estados e não é possível adicionar uma vez marcado, estará marcado para sempre. Os cinco conjuntos de estados diferentes que efetivamente caracterizam são:

$A = \{0, 1, 2, 4, 7\}$ $\quad$ $D = \{1, 2, 4, 5, 6, 7, 9\}$
$B = \{1, 2, 3, 4, 6, 7, 8\}$ $\quad$ $E = \{1, 2, 4, 5, 6, 7, 10\}$
$C = \{1, 2, 4, 5, 6, 7\}$

Deixamos o restante do preenchimento da tabela de transições $Dtran$ e mostramos o grafo de transições para o AFD resultante na Fig. 3.28. Deveria ser aparente que o AFD da Fig. 3.28 também aceita $(a|b)^*abb$ e possui um estado a menos. Discutiremos a questão da minimização do número de estados de AFDs na Seção 3.9. ■

3.7 DE UMA EXPRESSÃO REGULAR PARA UM AFN

Existem muitas estratégias para se construir um reconhecedor a partir de uma expressão regular, cada uma com suas próprias limitações e vantagens. Uma estratégia que tem sido usada em alguns programas de edição de texto e de construção de AFNs a partir de uma expressão regular é muito similar à computação do AFN numa cadeia de entrada usando os algoritmos 3.3 e 3.4 desta seção. Se o tempo de execução for essencial, podemos converter o AFN num AFD usando a construção de subconjuntos da seção anterior. Na Seção 3.9, exploraremos uma alter-

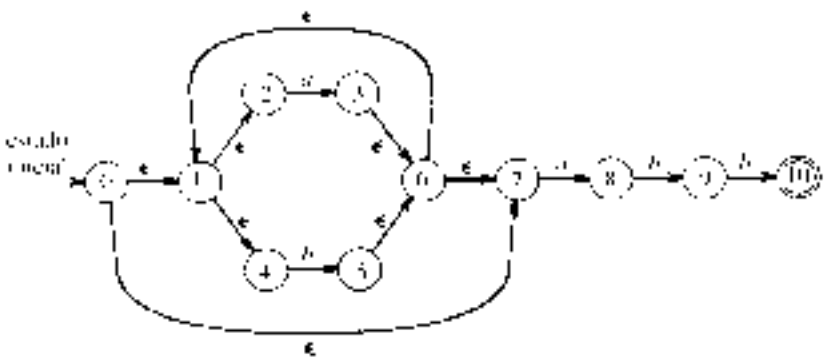


Fig. 3.27. AFN N para `(a|b)*abb`.

Estado	Símbolo de Entrada	
	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

Fig. 3.28 Tabela de transições D_{AFD} para o AFD

uma implementação de um AFD a partir de uma expressão regular. Se um AFN intermediário for explicitamente construído, isso pode envolver uma quantidade substancial de trabalho extra. Discutiremos sobre as barganhas de tempo-espaço na implementação de reconhecedores baseados em AFNs e AFDs.

Construção de um AFN a partir de uma Expressão Regular

Em seguida, um algoritmo para construir um AFN a partir de uma expressão regular. Existem muitas variantes desse algoritmo, mas aqui apresentamos uma versão simples que é fácil de implementar. O algoritmo é dirigido pela sintaxe na medida em que usa a estrutura sintática da expressão regular para guiar o processo de construção. As alternativas no algoritmo seguem as alternativas na definição de uma expressão regular. Primeiro mostramos como construir autômatos que reconhecem $\emptyset$, ε e qualquer símbolo no alfabeto. Em seguida, mostramos como construir autômatos para expressões contendo um operador de alternação, concatenação ou fecho de Kleene. Por exemplo, para a expressão $r_1 | r_2$ nós construímos um AFN combinando os AFNs para r_1 e r_2.

À medida que a construção prossegue, cada passo é ilustrado pelo menos dos possíveis estados e arestas do AFN resultante. Constrói-se a partir de uma expressão regular, passo a passo, um diagrama de estados, percorrendo a árvore de sintaxe abstrata de símbolos e de operadores encontrados na expressão regular.

Algoritmo 3.1. *Construção de Thompson.* Um AFN N a partir de uma expressão regular.

Entrada. Uma expressão regular r sobre um alfabeto Σ.

Saída. Um AFN N que aceita $L(r)$.

Método. Primeiro dividimos r em suas subexpressões constituintes. Em seguida, usando as regras (1) e (2) abaixo, construímos AFNs para cada um dos símbolos básicos em r (aqueles que se tornam um símbolo do alfabeto). Os símbolos básicos correspondem a ε e a $\emptyset$ na definição de uma expressão regular. É importante

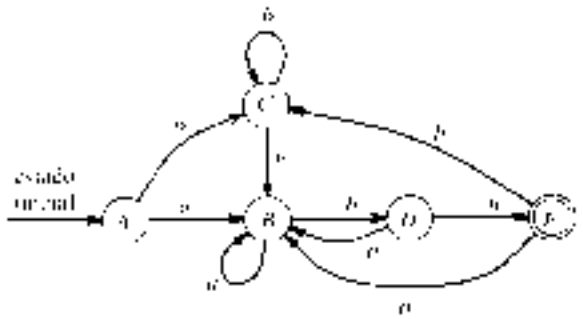


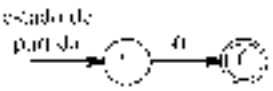
Fig. 3.29 Resultado da aplicação da construção de subexpressões à Fig. 3.22

compreender que r pode ocorrer várias vezes em r, mas para cada ocorrência é construído um AFN separado. Em seguida, juntamos os AFNs recém-criados combinando a regra (3) abaixo, até que obtenhamos o AFN para toda a expressão. Cada AFN intermediário produzido durante a construção corresponde a uma subexpressão de r e possui várias propriedades importantes: possui exatamente um estado final, nenhum lado entra no estado de partida e nenhum lado deixa o estado final.

1. Para ε construímos o AFN

Aqui, i é um novo estado de partida e f um novo estado de aceitação. Este AFN claramente reconhece ε.

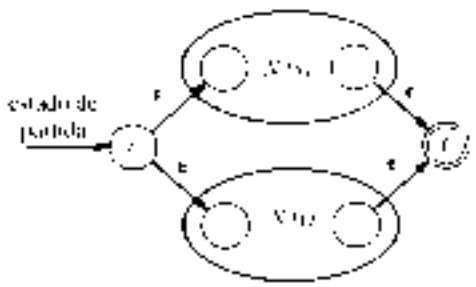
2. Para a em Σ, construímos o AFN



Aqui, i e f são novos estados, f um novo estado de aceitação. Esta máquina reconhece $\{a\}$.

3. Supondo que os AFNs $N(s)$ e $N(t)$ sejam AFNs para as expressões regulares s e t,

a) Para a expressão regular $s|t$, construímos o seguinte AFN composto $N(s|t)$:



Aqui, i é um novo estado de partida e f um novo estado de aceitação. Existe uma transição em ε a partir de i para os estados de partida de $N(s)$ e $N(t)$. Existe uma transição em ε a partir dos estados de aceitação de $N(s)$ e $N(t)$ para o novo estado de aceitação f. Os estados inicial e de aceitação de $N(s)$ e de $N(t)$ não são estados inicial e de aceitação de $N(s|t)$. Note-se que qualquer percurso de i para f precisa passar exclusivamente através de $N(s)$ ou de $N(t)$. Dessa forma, vemos que o AFN composto reconhece $L(s) \cup L(t)$.

b) Para a expressão regular st, construímos o AFN composto $N(st)$:

Os estados de partida de $N(r)$ e de $N(s)$ tornam-se, respectivamente, os estados de partida e de aceitação do AFN composto. O estado de aceitação de $N(r)$ é fundido ao estado de partida de $N(s)$, de sorte que todas as transições que originalmente saíam do estado de partida de $N(s)$ saem agora do estado de aceitação de $N(r)$. O novo estado resultante é, então, perder tanto o estado de partida de $N(r)$ quanto o de partida de $N(s)$. Um percurso de s para t passa ou por todos os estados de $N(r)$ ou, em seguida, através de $N(s)$, de tal forma que o rótulo do percurso seja uma cadeia em $L(r)L(s)$. Dessa forma, podemos ficar em estados de partida de $N(r)$ e de aceitação de $N(s)$, mas não podemos ir de s para t passando através de $N(r)$ uma ou mais vezes. O AFN composto reconhece, corretamente, $L(r)L(s)$.

d) Para uma expressão regular parenteseada, use os mesmos próprios $N(r)$ como o AFN.

A cada vez que construímos um novo estado, temos o cuidado de dar-lhe um nome distinto daqueles já existentes. Dessa maneira, dois estados de quaisquer AFN componentes não poderão ter o mesmo nome. Ainda que o mesmo símbolo apareça várias vezes em r, cada ocorrência desse símbolo gera um AFN separado com seus próprios estados.

Podemos verificar que cada passo da construção do Algoritmo 3.23 produz um AFN que reconhece a linguagem correta. Adicionalmente, a construção produz um AFN $N(r)$ com as seguintes propriedades:

1. $N(r)$ possui, no máximo, tantos estados quanto o dobro do número de símbolos e operadores em r. Isso segue do fato de que cada passo da construção cria no máximo dois novos estados.

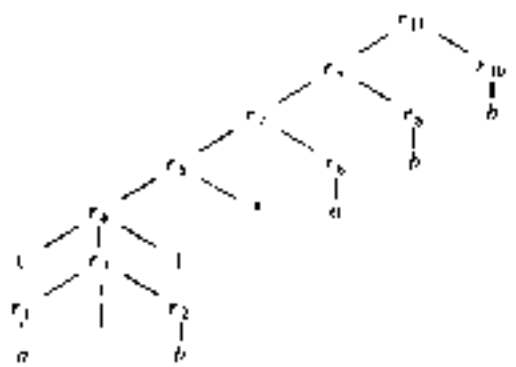


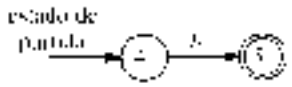
Fig. 3.38. Decomposição de $(a|b)^*a$.

2. $N(r)$ possui exatamente um estado de partida e um de aceitação. O estado de aceitação não possui transições para fora. Esta propriedade é válida igualmente para cada um dos autômatos constituintes.

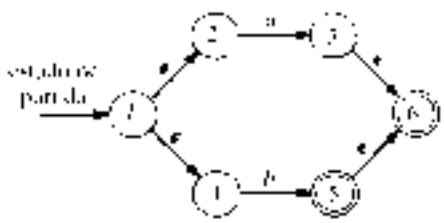
3. Cada estado de $N(r)$ possui ou uma transição para fora, rotulada com um símbolo de Σ, ou no máximo duas transições para fora, rotuladas com ϵ.

Exemplo 3.16. Vamos usar o Algoritmo 3.23 para construir $N(r)$ para a expressão regular $r = (a|b)^*a$. A Fig. 3.38 mostra uma árvore gramatical para r que é análoga às árvores gramaticais construídas para as expressões aritméticas na Seção 2.2. Para construirmos $N(r)$ para r, construímos os AFN

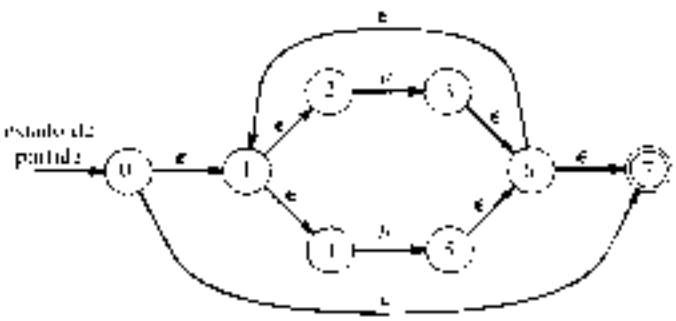
Para r_1 e r_2 temos



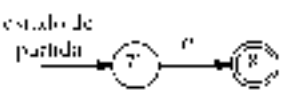
Podemos agora combinar $N(r_1)$ e $N(r_2)$ usando a regra da união para obter o AFN para $r_3 = r_1 | r_2$



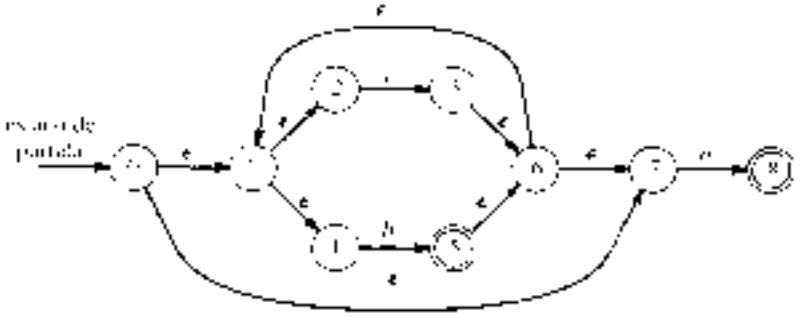
O AFN para r_4 é o mesmo que o para r_3. O AFN para r_5 é o seguinte:



O AFN para $r = r_5 a$



A fim de se obter o autômato para $r_5 a$, combinamos os estados 7 e 7, chamando o estado resultante de 7, por conter



Continuando dessa maneira, obtemos o AFN para $(a|b)^*abb$ mostrado na Fig. 3.37.

Simulação em Duas Pilhas de um AFN

Apresentamos agora um algoritmo que, dado um AFN N construído pelo Algoritmo 3.3 e uma cadeia de entrada x terminada em eof, determina se N aceita x. O algoritmo trabalha através de leitura de um caractere de cada vez em x, mantendo um conjunto de estados em que N poderia estar após ter lido a parte já vista da entrada. O algoritmo aproveita a vantagem de propriedades especiais do AFN construído pelo Algoritmo 3.3 para manipular cada conjunto de estados e cada transição em tempo proporcional a N. Pode-se implementar o algoritmo para rodar em tempo proporcional a N vezes o número de estados em N e x com comprimento constante.

Algoritmo 3.4. Simulação de um AFN.

Entrada. Um AFN N construído pelo Algoritmo 3.3 e uma cadeia de entrada x terminada com eof. Assumimos que o estado inicial, pelo caractere de fim de arquivo, eof, possui somente uma transição para o estado 0.

Saída. A resposta "sim" se N aceita x, "não" em caso contrário.

Método. Aplicar o algoritmo delineado na Fig. 3.38 para calcular D_{prox} a cada caractere de entrada. O algoritmo se refere a três funções que lidam com os conjuntos de estados e com a função de transição $move$ do AFN. A função $fecho\_vazio(s)$ empilha os estados que podem ser atingidos a partir do estado s através de zero ou mais transições ε. O algoritmo usa $fecho\_vazio(D_{atual})$ para obter os estados que podem ser atingidos a partir de D_{atual} através de zero ou mais transições ε. O algoritmo usa $fecho\_vazio(move(D_{atual}, a))$ para obter os estados que podem ser atingidos a partir de D_{atual} através de uma transição a seguida de zero ou mais transições ε. Quando todos os caracteres de entrada tiverem sido examinados, o algoritmo verifica se algum estado de aceitação está no conjunto de estados corrente, "sim" em caso contrário.

```
N = pré-processamento(N);
ε = procedimento_vazio();
empilhar(ε, eof, ε);
        N = fecho_vazio(estado inicial de N);
        a = próximo caractere;
fim
se N aceita a então
        retornar "sim"
senão retornar "não";
```

Fig. 3.38. Simulação de um AFN de Thompson.

Usamos uma pilha para conter o conjunto corrente de estados não-deterministas e uma pilha para armazenar o próximo conjunto de estados não-deterministas. Podemos usar o algoritmo da Fig. 3.26 para computar ε-fechos vazios. O conjunto de pilhas pode ser usado para determinar se um estado já foi colocado na pilha, de modo a não adicionar mais duas vezes a mesma. Uma vez que trabalhamos com apenas dois conjuntos de estados na mesma pilha, podemos colocar ambos os pares de estados lado a lado. Com cada estado também podemos armazenar duas transições para o próximo estado, de modo que cada estado possui no máximo dois sucessores. Os dois novos estados na transição a a partir de N contêm todos os estados de N. Como existem no máximo N estados no próximo conjunto de estados, podemos armazenar os estados de D_{prox} em uma pilha e usar um vetor indexado pelos estados de N para determinar em tempo constante se um estado já está na pilha.

Usamos também uma pilha para conter o conjunto corrente de estados não-deterministas e uma pilha para armazenar o próximo conjunto de estados não-deterministas. Podemos usar o algoritmo da Fig. 3.26 para computar ε-fechos vazios. O conjunto de pilhas pode ser usado para determinar se um estado já foi colocado na pilha, de modo a não adicionar mais duas vezes a mesma. Uma vez que trabalhamos com apenas dois conjuntos de estados na mesma pilha, podemos colocar ambos os pares de estados lado a lado. Com cada estado também podemos armazenar duas transições para o próximo estado, de modo que cada estado possui no máximo dois sucessores. Os dois novos estados na transição a a partir de N contêm todos os estados de N. Como existem no máximo N estados no próximo conjunto de estados, podemos armazenar os estados de D_{prox} em uma pilha e usar um vetor indexado pelos estados de N para determinar em tempo constante se um estado já está na pilha.

Exemplo 3.17. Seja N o AFN da Fig. 3.37 e suponha a cadeia de entrada $aabb$ seguida do caractere eof. O estado de partida é ε-fecho$(0) = \{0, 1, 2, 4, 7\}$. Ao ler a, os estados 3 e 8 são os únicos para os quais existe uma transição de 2 para N ou 4 para N. Portanto, $D_{prox} = \{3, 8\}$. Ao ler o segundo a, os estados 4 e 7 são substituídos por D_{prox} contendo $\{3, 8, 4, 7\}$. Ao ler b, D_{prox} contém $\{5, 6\}$. Ao ler o último b, D_{prox} contém $\{7, 8\}$. Como 8 é o estado de aceitação de N, a cadeia $aabb$ é aceita.

Note que o Algoritmo 3.4 realiza a construção de subconjuntos em tempo de execução. Por exemplo, comparando as transições acima com os estados do AFD da Fig. 3.24 construído a partir do AFN da Fig. 3.37, os conjuntos de estados iniciais e sucessores correspondem aos estados A e B do AFD.

Barganhas de Tempo-Espaço

Dada uma expressão regular e uma cadeia de entrada, temos agora dois métodos para determinar se a cadeia é uma sentença gerada pela expressão. O Algoritmo 3.3 seguido pela simulação de um AFN na parte de cima da Fig. 3.38 pode ser dividido em duas fases. A primeira constrói um AFN N a partir de r. Esta construção pode ser feita em tempo $O(r)$ e o número de estados de N é no máximo o dobro do número de caracteres (operandos e operadores) em r. A segunda fase usa N para determinar se a cadeia de entrada x pode ser gerada por r. Usando a representação de matriz de transições para N, o tempo de processamento de cada caractere de x é $O(n)$, onde n é o número de estados de N. Portanto, o tempo total para reconhecer x é $O(|x|n)$. Assim, o tempo total para reconhecer x é proporcional a $|r|$ vezes $|x|$.

Uma segunda abordagem é a de construir um AFD a partir de uma expressão regular r através da aplicação da construção de Thompson para obter um AFN e, em seguida, usar a construção de subconjuntos. Algoritmo 3.5 para o AFN é talvez uma implementação que evita a construção explícita de um AFN da Seção 3.7. Dependendo da linguagem de transição com a tabela de transições, podemos usar o Algoritmo 3.1 para simular o AFD resultante. O tempo para reconhecer uma cadeia de entrada x pode ser limitado por $|x|$, independente do número de estados do AFD. Porém, o tempo de construção pode ser muito maior. Veremos mais adiante que, em alguns editores de texto, é preferível usar a primeira abordagem para compilar expressões regulares quando a cadeia alvo é razoavelmente curta.

Uma segunda abordagem é a de construir um AFD a partir de uma expressão regular r através da aplicação da construção de Thompson para obter um AFN e, em seguida, usar a construção de subconjuntos. Algoritmo 3.5 para o AFN é talvez uma implementação que evita a construção explícita de um AFN da Seção 3.7. Dependendo da linguagem de transição com a tabela de transições, podemos usar o Algoritmo 3.1 para simular o AFD resultante. O tempo para reconhecer uma cadeia de entrada x pode ser limitado por $|x|$, independente do número de estados do AFD. Porém, o tempo de construção pode ser muito maior. Veremos mais adiante que, em alguns editores de texto, é preferível usar a primeira abordagem para compilar expressões regulares quando a cadeia alvo é razoavelmente curta.

AUTÔMATO	ESPAÇO	TEMPO
NFA	$O(\quad)$	$O(\quad)$
DFA	$O(2^{\quad})$	$O(\quad)$

Fig. 3.32. Espaço e tempo necessários para reconhecer expressões regulares.

reconhecimento de padrões que percorrem arquivos de texto procurando padrões de expressões regulares. Uma vez que o autômato finito tenha sido construído, a procura pode proceder muito rapidamente, e então esse enfoque é vantajoso quando a cadeia alvo é muito longa.

Existem, entretanto, certas expressões regulares cujo menor AFD possui um número de estados cujo tamanho é uma exponencial do tamanho da expressão regular. Por exemplo, a expressão regular $(a|b)^*a(a|b)(a|b)\ldots(a|b)$, onde existem $n-1$ $(a|b)$'s no final, não possui um AFD com menos de 2^n estados. Essa expressão regular denota aquele conjunto de cadeias de a's e b's na qual o n-ésimo símbolo a partir do extremo da direita é um a. Não é difícil provar que qualquer AFD para esta expressão precisa controlar os últimos n caracteres que examinou na entrada de uma forma ou de outra. pode-se fornecer uma resposta exata. Conseqüentemente, são necessários pelo menos 2^n estados para se controlar todas as possíveis seqüências de n a's e b's. Felizmente, expressões como essa não ocorrem muito frequentemente em aplicações de análise léxica, mas existem aplicações onde expressões similares certamente ocorrerão.

Um recurso criativo consiste em projetar um AFD, mas evitar construir toda a tabela de transições antes do tempo através de uma técnica chamada "avaliação preguiçosa de transições". Aqui, as transições são computadas em tempo de execução, isto é, uma transição a partir de um dado estado e com um símbolo não determinado ainda que seja efetivamente necessitada. As transições computadas são armazenadas num cache. A cada vez que uma transição precisa ser realizada, o cache é consultado. Se a transição não estiver lá, então ela é computada e armazenada no cache. Se o cache encher, podemos apagar algumas transições previamente computadas a fim de abrir espaço para a nova transição.

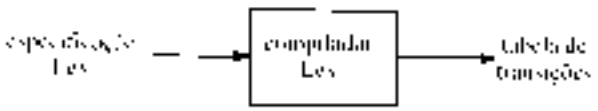
A Fig. 3.32 sumariza o pior caso de exigência de espaço e tempo para se determinar se uma cadeia de entrada está na linguagem denotada por uma expressão regular r, usando reconhecedores que foram construídos a partir de autômatos determinísticos e não-determinísticos. A técnica "preguiçosa" combina as exigências de espaço e tempo do AFN com as de tempo da abordagem do AFD. Sua exigência de espaço é o tamanho da expressão regular mais o tamanho do cache. Seu tempo de execução observado é quase tão bom quanto aquele de um reconhecedor determinístico. Em algumas aplicações, a técnica "preguiçosa" é considerada a mais rápida do que o enfoque do AFD, porque o custo extra de despesa adicional computando as transições de estados que jamais serão usadas.

3.8 O PROJETO DE UM GERADOR DE ANALISADORES LÉXICOS

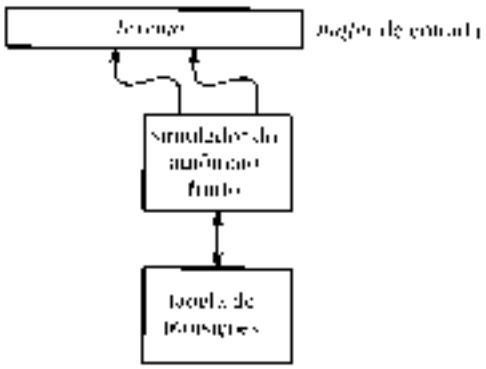
Nesta seção, consideramos o projeto de uma ferramenta de software que constrói automaticamente um analisador léxico a partir de um programa na linguagem Lex. Apesar de existirem vários métodos e de nenhum ser precisamente idêntico àquele usado pelo comando Lex do sistema UNIX, referimo-nos a esses programas para construir analisadores léxicos como compiladores Lex.

Assumimos que temos uma especificação de um analisador léxico da forma

$$\begin{aligned} p_1 \quad & \{ \text{ação}_1 \} \\ p_2 \quad & \{ \text{ação}_2 \} \\ & \cdots \\ p_n \quad & \{ \text{ação}_n \} \end{aligned}$$



(a) Compilador Lex.



(b) Analisador léxico esquematizado

Fig. 3.33. Criação de um analisador Lex.

onde, como na Seção 3.5, cada padrão p_i é uma expressão regular e cada ação_i é um fragmento de um programa que deve ser executado sempre que um lexema reconhecido por p_i for encontrado na entrada.

Nosso problema é de construir um reconhecedor que procure por lexemas no *buffer* de entrada. Se mais de um padrão for reconhecido, o reconhecedor deverá escolher o mais longo lexema encontrado. Se existirem dois ou mais padrões que reconheçam o lexema mais longo, o primeiro padrão da lista do compilador Lex será escolhido.

Um projeto típico é um modelo natural em torno do qual se pode construir um analisador léxico, tal como construído por nosso compilador Lex, possui a forma ilustrada na Fig. 3.33(b). Lexemas de entrada são armazenados para o momento num buffer de entrada. O ponteiro *avançar* move-se através do buffer de entrada, como discutido na Seção 3.2. O compilador Lex constrói uma tabela de transições para um autômato finito a partir de padrões de expressões regulares na especificação Lex. O autômato finito em si é uma rotina que simula o autômato finito. Cada vez que uma nova transição é necessária, o simulador do autômato finito examina a tabela de transições para proceder pelos padrões de expressões regulares no *buffer* de entrada.

O resto desta seção mostra que a implementação de um compilador Lex pode ser baseada em um autômato determinístico, que não é muito determinístico. Ao fim da última seção, vimos que a tabela de transições de um AFN para um padrão de expressão regular pode ser consideravelmente menor do que a de um AFD, mas que o AFD possui a desejada vantagem de ser capaz de reconhecer palavras muito rapidamente (tempo n para um AFN).

Reconhecimento de Padrões Baseado em AFN's

Um método é o de construir a tabela de transições de um autômato finito não-determinístico N para o padrão composto $p_1|p_2|\cdots|p_n$. Isto pode ser feito primeiro criando-se um AFN $N(p_i)$ para cada padrão p_i, usando o Algoritmo 3.3 e, em seguida, adicionando-se um novo estado de partida s_0, e finalmente ligando-se s_0 ao estado de partida de cada $N(p_i)$ através de uma transição ε, como ilustrado na Fig. 3.34.

Para simular este AFN, podemos usar uma modificação do Algoritmo da Fig. 3.4. A modificação assegura que o AFN combinará aquele prefixo mais longo da entrada que seja reconhecido por um padrão. No AFN combinado N, existe um estado de aceitação para cada padrão p_i. As simulações do AFN usando o Algoritmo 3.4 avança

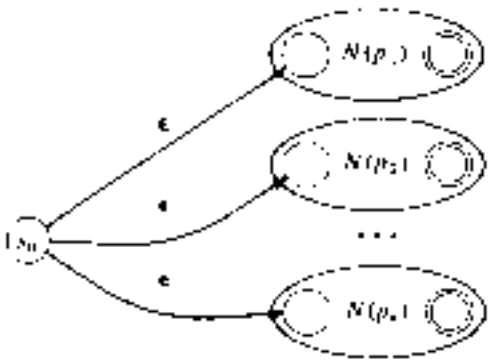


Fig. 3.34. AFN gerado a partir de uma especificação Lex

funções a seguir são de conjuntos de estados e projetar o AFN com forma de possibilitar-nos examinar cada caractere de entrada. Mesmo se estivéssemos interessados em obter o estado que corresponde ao estado de aceitação para obtermos a correspondência mais longa precisaríamos continuar a simular o AFN até que ele atingisse a *terminação*, isto é, até que o conjunto de estados a partir do qual não existam transições no símbolo corrente de entrada.

Presumimos que a especificação Lex seja projetada de uma forma que um programa cujo fluxo de entrada não possa prosseguir além de um caractere de entrada, sempre que o AFN venha a atingir a terminação. Por exemplo, cada computador coloca alguma restrição no comprimento de um identificador: violações desse limite serão detectadas quando o *buffer* de entrada estourar a capacidade de um programa escrito para usá-lo.

Para chegar ao reconhecimento correto, fazemos duas modificações ao Algoritmo 3.4. Primeiro, sempre que adicionamos um estado de aceitação ao conjunto corrente de estados, registramos a posição corrente de entrada e o padrão p_i correspondente a esse estado de aceitação. Se o conjunto contiver estados de aceitação para mais de um padrão, então estamos diante de uma ambiguidade e preferimos a especificação de Lex que vem primeiro. Segundo, continuamos fazendo transições até encontrarmos a terminação. Ao final, retornamos a apontar adiante para a posição marcada pelo último reconhecimento correto. O padrão que realizou esse reconhecimento identifica o token encontrado e lexema associado obtido é a cadeia entre os apontadores de início e de fim.

Usualmente, a especificação Lex é tal que algum padrão possibilite reconhecimento em qualquer circunstância. Se nenhum padrão o fizer, então, temos uma condição de erro para a qual nenhuma providência lexical é tentada. Talvez devêssemos gerar um erro léxico para alguma rotina *default* de recuperação de erros.

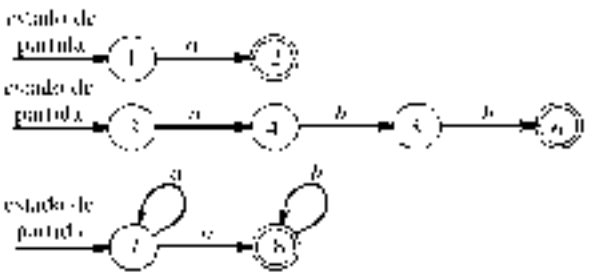
Exemplo 3.18. Um único exemplo ilustrará as idéias acima. Suponhamos o seguinte programa Lex consistindo em três expressões regulares e uma ação de função tal qual

a	{ /* ação que não é mostrada aqui */ }
abb	{ }
a^+b	{ }

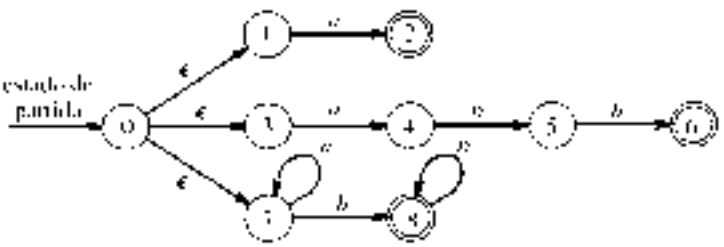
Os três padrões acima são reconhecidos pelos autômatos da Fig. 3.35(a). Simplificamos o diagrama, unificando em alguns pontos os estados que teriam sido produzidos pelo Algoritmo 3.3. Considerando acima, podemos converter os AFNs da Fig. 3.35(a) em único AFN combinado. Mostrado na Fig. 3.35(b).

Vamos considerar o comportamento de AFN sobre a cadeia de entrada *aabbb*, usando as definições do Algoritmo 3.4. A Fig. 3.36 mostra os conjuntos de estados e padrões que recombinam à medida que cada caractere de entrada *aabbb* é processado. Essa figura mostra que o conjunto inicial de estados é $\{0, 1, 3, 7\}$. Os estados 1, 3 e 7 têm cada um uma transição em *a* para os estados 2, 4 e 8, respectivamente. O estado 2 é o de aceitação para o primeiro padrão, registramos o fato de que o primeiro padrão reconhecido corresponde após a leitura do primeiro *a*.

Entretanto, existe uma transição a partir do estado 7 para o estado 8 no segundo caractere de entrada, que prosseguimos confirmar realizando as transições. Há ainda uma transição a partir do estado 8 para o 8 no caractere de entrada *b*. O estado 8 é o estado de aceitação para o terceiro padrão. Uma vez que atingimos o estado 8, não existem transições possíveis para o próximo caractere de entrada e nesse caso atingimos a terminação. Como o último reconhecimento ocorreu depois da leitura do terceiro caractere de entrada, relatamos que o terceiro padrão reconheceu o lexema *aab*. ∎



(a) AFN para a, abb e a^+b



(b) AFN combinado

Fig. 3.35. AFN para reconhecer três diferentes padrões

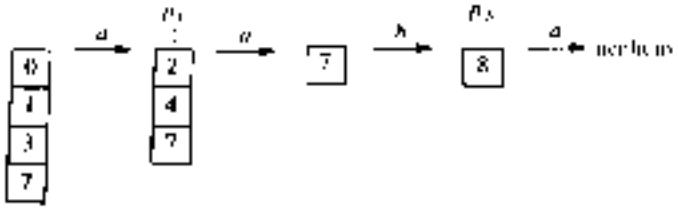


Fig. 3.36. Sequência de estados ativos no processamento da cadeia *aabba*.

O papel da função `escolha_padrão` na especificação Lex é como se segue. Quando uma instância de cada padrão é reconhecida, a função `escolha_padrão` retorna o número do padrão escolhido. Note-se que ação não é executada somente porque o AFN entra num estado que inclui um estado de aceitação para o padrão. A ação somente é executada quando se sabe que o padrão reconhecido é o mais longo.

AFDs para Analisadores Léxicos

Um outro enfoque para a construção de um analisador léxico é o de usar um AFD para reconhecer o reconhecimento de padrões. A única mudança é estrutural: em vez de encontrarmos os padrões de reconhecimento adequados, a simulação é completa, incluindo a simulação não-adequada do AFN pelo escritor. Quando convertemos um AFN para um AFD usando os métodos de construção de subconjuntos do Algoritmo 3.2, podemos ter vários estados de aceitação em cada subconjunto de estados não-determinísticos. Em tal situação, o estado de aceitação correspondente ao padrão listado primeiro especificação Lex possui prioridade. Uma outra situação em AFNs é ilustrada pela ação que precisamos fazer ao detectarmos reconhecedores de palavras-chave e identificadores. Normalmente, preferimos a palavra-chave, a menos que nenhuma palavra-chave esteja presente, caso em que preferimos o identificador. Para encontrar o lexema reconhecido, retornamos à última posição de entrada na qual o AFD entrou num estado de aceitação.

Exemplo 3.19. Se convertermos o AFN da Fig. 3.35 para um AFD, obteremos a tabela de transições da Fig. 3.37, onde os estados do AFD foram dados por listas de estados do AFN. A última coluna da Fig. 3.37 indica as ações recomendadas associadas àquele estado do AFD. Por exemplo, dentre os estados 2, 4 e 7 do AFN, somente 2 é um estado de aceitação e o reconhecedor de padrões do autômato para a expressão regular *a* na Fig. 3.35(a). Consequentemente, o estado 247 do AFD recebe a ação padrão 1.

Note-se que a cadeia *aabba* é reconhecida por todos os padrões, pois é levada aos estados 2, 4 e 8 do AFN. O estado 68 do AFD, contendo a lista de transições incluindo os estados de aceitação para o AFN, informa que *abba* é reconhecida pela terceira regra de tradução de nossa especificação Lex. Também anunciamos que *abba* foi encontrada no estado 68 do AFD.

Como vimos, a cadeia *aabba* leva o AFD a entrar nos estados sucessivos para a simulação do AFN mostrada na Fig. 3.36. Observe-se que os seguintes exemplos ilustram a entrada dos estados do AFD da Fig. 3.37 ao longo do estado 0137. A entrada a um par no estado 247 faz com que a próxima entrada *b* prossiga até o estado 58, e a entrada *a* mantém o autômato no estado 247. Assumimos, para o exemplo, a terminação, progredindo através dos estados 247, na segunda 247 e então 58. O último desses estados inclui o estado de aceitação 8 da Fig. 3.35(a). Quando, então, encontramos, no estado 58, o AFD informa o padrão 3, *abba*. Uma reconhecedora seleciona o padrão da entrada que levou ao estado 58 mais recentemente.

Estados	Símbolos de Entrada		Padrão Anunciado
	a	*b*	
0137	247	8	nenhum
247	7	58	1
8		8	nenhum
7	7	8	nenhum
58	—	68	3
68	—	8	3

Fig. 3.37. Tabela de transições para um AFD.

Implementando o Operador *lookahead*

Relembre-se da Seção 3.4 que o operador *lookahead* é necessário em algumas situações, dado que o padrão que denota um *token* particular pode precisar descrever algum contexto após o lexema efetivo. Quando convertemos um padrão com / para um AFN, podemos tratar o símbolo / como se fosse ε, pois o reconhecedor léxico irá prosseguir pela entrada, tentando verificar uma cadeia denotada por esta expressão regular for reconhecida no *lookahead*. Porém, o final do lexema não é a posição do estado de aceitação do AFN. Ao invés, é a última ocorrência de estado nesse AFN que tenha uma posição correspondente à /.

Exemplo 3.20. O AFN que reconhece o padrão 1 dado no Exemplo 3.10 é mostrado na Fig. 3.38. O estado 6 indica a presença da palavra-chave `if`, lembrando-se de que encontramos o estado 6 após encontrarmos `if` seguido de zero ou mais ocorrências de estado 2.

3.9 OTIMIZAÇÃO DE RECONHECEDORES DE PADRÕES BASEADOS EM AFDs

Nesta seção, apresentamos três algoritmos que têm sido usados para implementar e otimizar reconhecedores de padrões construídos a partir de expressões regulares. O primeiro algoritmo é adequado para inclusão num compilador Lex, porque constrói um AFD diretamente a partir de uma expressão regular, sem construir um AFN intermediário no longo caminho.

O segundo algoritmo minimiza o número de estados de qualquer AFD determinístico. Pode ser usado para reduzir o tamanho de reconhecedores de padrões baseados em AFDs. O algoritmo é eficiente, sem tempo de execução $O(n \log n)$, onde n é o número de estados do AFD. O terceiro algoritmo produz representações muito compactas para as tabelas de transições de AFDs, porém pode tornar a simulação do AFD mais lenta. Especificamente, ele pode ser usado para produzir representações rápidas para manipular tabelas de transições esparsas, isto é, tabelas em que a maioria das entradas é vazia. Uma tabela bidimensional

Estados Importantes de um AFN

Vamos chamar de *importante* um estado de um AFN que tenha uma transição fora. A construção de subconjuntos na Fig. 3.25 cria somente

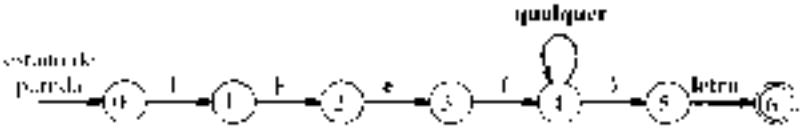


Fig. 3.38. AFN reconhecendo a palavra-chave `if` e `id`.

os estados importantes de um subconjunto I quando detectamos a ocorrência de um símbolo específico x. Especificamente, D é o conjunto de estados que são alcançados a partir de I e exatamente um símbolo x. Os conjuntos construídos dessa forma são chamados de **conjuntos de Dmarker**. Como a construção dos subconjuntos pode ser modificada de modo que os conjuntos de Dmarker sejam calculados simultaneamente, vamos nos referir a esses conjuntos como **conjuntos de Dmarker**. Os conjuntos de Dmarker incluirão estados de aceitação do AFN.

Quando a construção de subconjuntos é aplicada a um AFN obtido pelo Algoritmo 3.5 a partir de uma expressão regular, podemos explorar as propriedades especiais do AFN para calcular os conjuntos de Dmarker. A construção concentra-se nos estados importantes do AFN com os símbolos correspondentes da expressão regular. A construção de Thompson modela um estado importante exatamente quando um símbolo concatenado a uma expressão regular. Por exemplo, os estados importantes serão construídos para cada ocorrência do símbolo a.

Sobrepondo o AFN resultante possui exatamente um estado de aceitação, mas esse estado não é importante porque não possui transições para fora. Pela concatenação de um marcador único $\#$ ao final de uma expressão regular, damos ao estado de aceitação do AFN para $r\#$ uma importância $\#$. Em outras palavras, usando a expressão regular aumentada $(r)\#$, podemos esperar que os estados de aceitação produzidos quando a construção de subconjuntos pressupõe aplicada ao AFN estejam associados ao padrão r e tenham especificidade de aceitação.

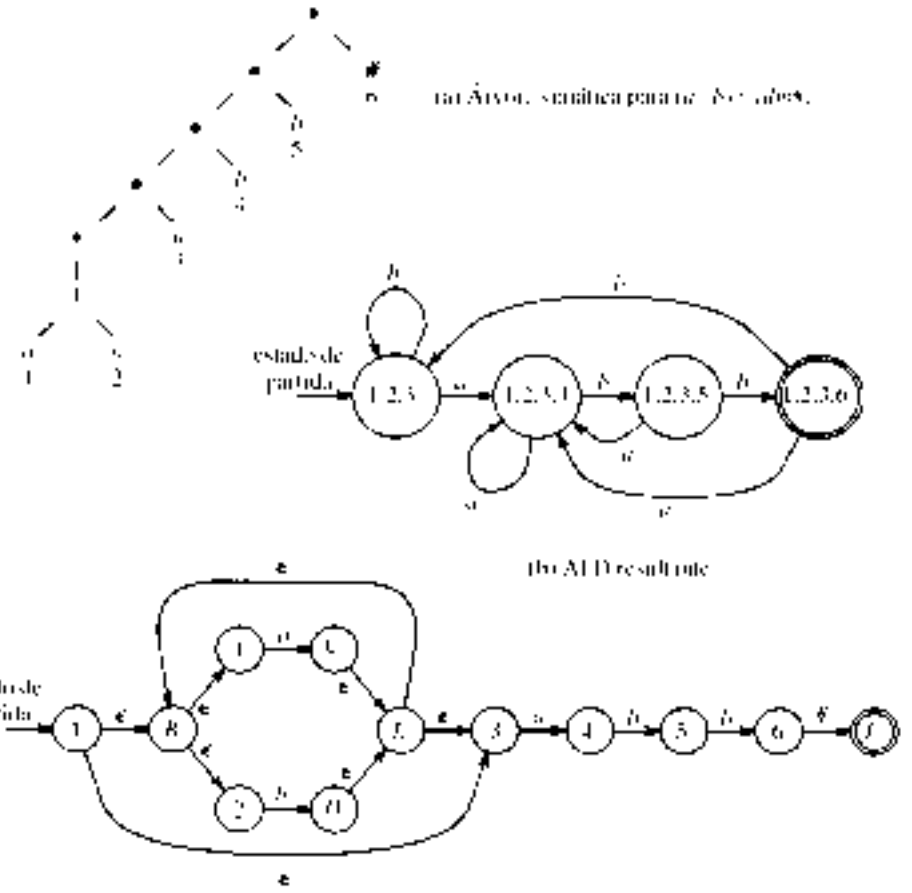
Representamos uma expressão regular aumentada $r\#$ por uma árvore sintática com os símbolos básicos nas folhas e os operadores nos nós internos. Rotulamos cada folha com um único símbolo ou com um conjunto de símbolos, ou com ε, para indicar que a folha é ε. Cada nó interno é rotulado por um operador, concatenação ou repetição, respectivamente. A Fig. 3.39 (a) mostra a árvore sintática para a expressão regular aumentada $(a|b)^*abb\#$. Essa árvore sintática para uma expressão regular pode ser construída da mesma maneira que uma árvore sintática para uma expressão aritmética, como discutimos na Seção 2.5 do Capítulo 2.

As folhas na árvore sintática para uma expressão regular são rotuladas por símbolos de alfabeto ou por ε. Para cada folha rotulada por a, armazenamos um único inteiro que referencia o nó, ou posição, da folha. Também, como a posição de seu símbolo. Um símbolo pode ter várias posições. As posições são numeradas, a partir de 1, da esquerda para a direita na árvore sintática da Fig. 3.39 (a). Os estados numerados no AFN da Fig. 3.36 correspondem às posições da árvore sintática da Fig. 3.39 (a). Não é uma mera coincidência que esses conjuntos estados importantes do AFN. Os estados não importantes são designados por letras maiúsculas na Fig. 3.39 (a).

O AFD na Fig. 3.39 (b) pode ser obtido a partir do AFN na Fig. 3.36 aplicando uma construção de subconjuntos e identificando, entre os estados resultantes, os estados importantes. A identificação resulta da construção de um estado para cada um dos conjuntos na computação na Fig. 3.20.

De uma Expressão Regular para um AFD

Nesta seção, mostramos como construir um AFD diretamente a partir de uma expressão regular expandida $r\#$. Começamos construindo uma árvore sintática T para $r\#$ e, em seguida, computamos quatro funções: *anulável*, *primeira-pos*, *última-pos* e *seguinte-pos*, através de travessias na árvore T. Finalmente, constrói-se um AFD, partindo de *seguinte-pos*. As funções *anulável*, *primeira-pos* e *última-pos* são



(a) Árvore sintática para $(a|b)^*abb\#$.

(b) AFD resultante

(c) AFN sem ε-arestas.

Fig. 3.39. AFD e AFN construídos a partir de $(a|b)^*abb\#$

de ligadas sobre os nós da árvore sintática e usadas para computar *primeira_pos*, que é definida somente para os nós de posições.

Relembrando, os pontos de controle importantes de um AFN são as posições nas folhas na árvore sintática da expressão regular. Pedimos, então, uma função na construção do AFN através da modelagem de um AFD cujos estados correspondem aos conjuntos de posições nessa árvore. As transições do AFN representam alguma estrutura razoavelmente complexada das posições e, em particular, escolhemos a informação correspondente quando uma posição pode ser seguida uma outra. Dessa, cada símbolo numa cadeia de entrada pode, um AFD pode ser reconhecido por certas posições. Um símbolo de entrada pode somente ser reconhecido pelas posições naquele estado, mas existem toda posição que tenha uma para interessa somente reconhecer uma ocorrência particular de um dos símbolos de entrada.

A função de uma posição reconhecendo um símbolo de entrada será definida em termos da função *pos_seguinte* sobre as posições da árvore sintática. Se p for uma posição, então *pos_seguinte*(p) é o conjunto de posições q tal que existe alguma cadeia de entrada $\ldots ab \ldots$ em que a corresponde a esta ocorrência de a e esta ocorrência de b corresponde a q.

Exemplo 3.21. No AFN da Fig. 3.39 tem-se *pos_seguinte*$(1) = \{1, 2, 3\}$. A argumentação para tal é que se existir cadeia com uma ocorrência correspondente à posição 1, então acertamos de ver uma ocorrência de a (não necessariamente b). Podemos, em seguida, examinar a primeira posição de cada ocorrência de a ou b, o que explica por que 1 e 2 estão em *pos_seguinte*(1). Podemos também, em seguida, examinar a primeira posição do que segue a ocorrência de a, isso é, a posição 3.

A modo de computar a função *pos_seguinte*, necessitamos saber em que posições podem reconhecer o primeiro ou último símbolo de uma cadeia gerada por uma dada subexpressão de uma expressão regular. Tal informação foi usada, informalmente, no exemplo 3.21. Se r é uma tal subexpressão, cada posição que pode ser a primeira em uma cadeia gerada por r é chamada uma *primeira posição* de r. Similarmente, se r é uma subexpressão, cada primeira posição de r segue cada última posição de r.

A definição da árvore sintática de uma expressão regular, definimos uma função *primeira_pos*(n), que fornece o conjunto de posições que podem reconhecer o primeiro símbolo de uma cadeia gerada por uma subexpressão enraizada em n. Similarmente, definimos uma função *última_pos*(n) que fornece o conjunto de posições que podem reconhecer o último símbolo de uma tal cadeia. Por exemplo, se n é a raiz de toda a árvore na Fig. 3.39 tem-se *primeira_pos*$(n) = \{1, 2, 3\}$ e *última_pos*$(n) = \{6\}$. Mostraremos, em breve, como computar essas funções.

Para computar *primeira_pos* e *última_pos*, precisamos saber que nós sintáticos têm as subexpressões que geram linguagens que incluem a cadeia vazia. Tais nós são chamados *anuláveis* e definimos *anulável*(n) como sendo verdadeiro se, e somente se, n for um nó anulável e falso em caso contrário.

Precisamos agora fornecer as regras para computar as funções *anulável*, *primeira_pos* e *última_pos*. Para cada uma dessas três funções, temos uma regra básica que nos diz a respeito de um símbolo básico e três regras indutivas que nos permitem determinar o valor de uma função para um nó, baseando-se nos valores de seus filhos. Em cada caso, as regras indutivas correspondem aos três operadores de união, concatenação e fechamento. As regras para *anulável* e *primeira_pos* são dadas na Fig. 3.40. As regras para *última_pos* são as mesmas que as regras para *primeira_pos*, mas com c_1 e c_2 invertidos, e não são mostradas.

A primeira regra para *anulável* estabelece que se n é uma folha rotulada ε, então *anulável*(n) é certamente verdadeira. A segunda regra estabelece que se n é uma folha rotulada por um símbolo de alfabeto então *anulável*(n) é falsa. Nesse caso, cada folha corresponde a um único símbolo de cadeia e, consequentemente, não pode gerar ε. A última regra para *anulável* estabelece que se n é um nó representando filhos c_1 e c_2, então *anulável*(n) é verdade se, e somente se, *anulável*(c_1) e *anulável*(c_2) forem verdade, porque o fechamento de uma expressão gera uma linguagem que inclui ε.

Como um outro exemplo, a quarta regra para *primeira_pos* estabelece que se n for um nó com filhos c_1 e c_2, onde c_1 filho à direita de c_1, e se *anulável*(c_1) for verdadeiro, então

$$primeira\_pos(n) = primeira\_pos(c_1) \cup primeira\_pos(c_2)$$

de outra forma, *primeira_pos*$(n) =$ *primeira_pos*(c_1). O que essa regra diz é que se uma expressão gera ε, então as primeiras posições de "alternativas" são também primeiras posições de n; de outra forma somente as primeiras posições de c_1 são as primeiras posições de n. A seguinte tabela dá as regras restantes sobre *anulável* e *primeira_pos*, sem provas.

A função *pos_seguinte* nos diz que posições podem seguir uma posição na árvore sintática. Duas regras definem todas as formas em que uma posição pode ser seguida uma outra.

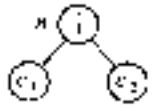
NÓ n	*anulável*(n)	*primeira_pos*(n)
n é uma folha rotulada ε	verdadeiro	$\varnothing$
n é uma folha rotulada com posição i	falso	$\{i\}$
	*anulável*(c_1) **ou** *anulável*(c_2)	*primeira_pos*$(c_1) \cup$ *primeira_pos*(c_2)
	anulável(c_1) **e** *anulável*(c_2)	**se** *anulável*(c_1) **então** *primeira_pos*$(c_1) \cup$ *primeira_pos*(c_2) **senão** *primeira_pos*(c_1)
	verdadeiro	*primeira_pos*(c_1)

Fig. 3.40. Regras para computar *anulável* e *primeira_pos*.

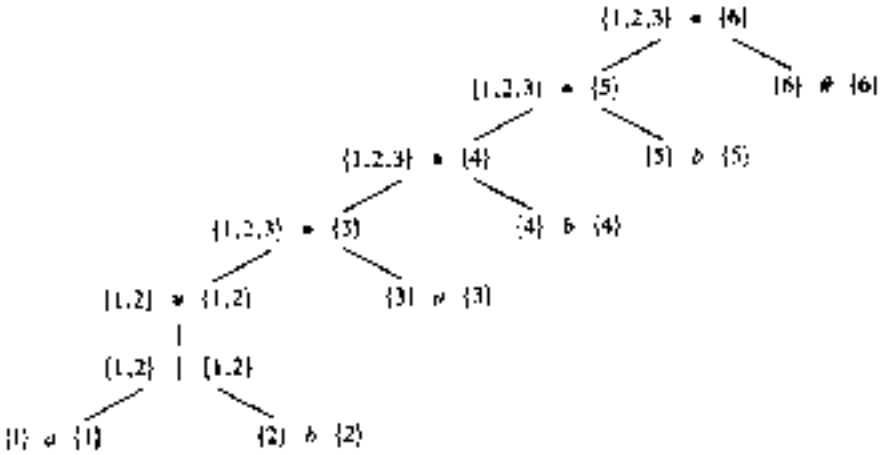


Fig. 3.41 primeira_pos e última_pos para o nó raiz e seus filhos da árvore sintática para (a|b)*abb#.

Se i pertence a *última_pos*(n), então todas as posições em *primeira_pos*(n) estão em *pos_seguinte*(i).

Se i é uma posição em *última_pos*(n), então todas as posições em *primeira_pos*(n) estão em *pos_seguinte*(i).

Se *primeira_pos* e *última_pos* forem computados para cada nó, *pos_seguinte* de cada posição pode ser calculado rapidamente se uma travessia em profundidade da árvore sintática

Exemplo 3.22. A Fig. 3.41 mostra os valores de *primeira_pos* e de *última_pos* em todos os nós da árvore da Fig. 3.40 (a|b)*abb#. *primeira_pos* do nó raiz aparece à esquerda do nó e *última_pos* à direita. Por exemplo, *primeira_pos* do nó à folha mais à esquerda é {1}, pois essa folha está rotulada com a posição 1. Similarmente, *primeira_pos* da segunda folha é {2}, pois essa folha está rotulada com a posição 2. Pela terceira regra na Fig. 3.40, *primeira_pos* do pai de ambas é {1, 2}.

A única folha com rótulo não anulável é a com a posição 6. Consequentemente, pela primeira regra, *última_pos* para o pai desse nó naquele momento é {6}. O nó rotulado | é a união de {1, 2} e {3}, que são primeira_pos de seus filhos à esquerda e à direita. Por outro lado, a concatenação rotulada • à direita para *última_pos* desse nó, dado que a folha da posição 3 não é anulável. Consequentemente, o pai do nó esquerdo possui *última_pos* contendo somente 3.

Vamos agora computar *pos_seguinte* de baixo para cima para cada um dos nós da árvore sintática da Fig. 3.41. No nó rotulado com a primeira • acima, temos $i = 1$ e 2, *pos_seguinte*(i) é o *primeira_pos* (3), usando a regra 2). Ao nível do pai do nó esquerdo, adicionamos 3 a *pos_seguinte*(1) e a *pos_seguinte*(2), usando a regra 1). Ao próximo nível, ao nó esquerdo, adicionamos 4 a *pos_seguinte*(3) usando a regra 1). Aos próximos dois nós, ele adicionamos 5 a *pos_seguinte*(4) e 6 a *pos_seguinte*(5) usando a mesma regra. Isso completa a construção de *pos_seguinte*. A Fig. 3.42 sumariza *pos_seguinte*.

Nó i	*pos_seguinte*(i)
1	{1,2,3}
2	{1,2,3}
3	{4}
4	{5}
5	{6}
6	—

Fig. 3.42. A função *pos_seguinte*.

Pode-se ilustrar a função *pos_seguinte* através de um grafo orientado com um nó para cada posição e um lado direcionado do nó para o nó i se i estiver em *pos_seguinte*(n). A Fig. 3.43 mostra esse grafo orientado para *pos_seguinte* da Fig. 3.42.

É interessante notar que esse diagrama vira-se tornar um AFN sem as transições-ε para a expressão regular em questão se

1. Fizéssemos com que todas as transições em *primeira_pos* do nó raiz fossem estados de partida.
2. Rotulássemos cada lado emanado de i pelo símbolo a posição i.
3. Fizéssemos a posição associada com # ser o único estado de aceitação.

Não deveria haver surpresa ao ver se pudéssemos converter o grafo *pos_seguinte* num AFD utilizando a construção de subconjuntos. A construção inteira pode ser concretizada através das posições usando-se o algoritmo seguinte

Algoritmo 3.7. Construção de um AFD a partir de uma expressão regular r.

Entrada. Uma expressão regular r.

Saída. Um AFD D que reconhece $L(r)$.

Método.

1. Construir uma árvore sintática T para a expressão regular aumentada $(r)\#$, onde # é um marcador único de fim, anexado ao fim de r.
2. Construir as funções *anulável*, *primeira_pos*, *última_pos* e *pos_seguinte* através de uma travessia em profundidade (*depth-first*) de T.
3. Construir $Estados$, o conjunto de estados de D, e $Dtran$, a tabela de transições para D, através do procedimento na Fig. 3.44. Os estados em $Estados$ são conjuntos de posições, inicialmente, cada estado está "não-marcado", e um estado se torna "marcado" exatamente antes de considerarmos suas transições para fora. O estado

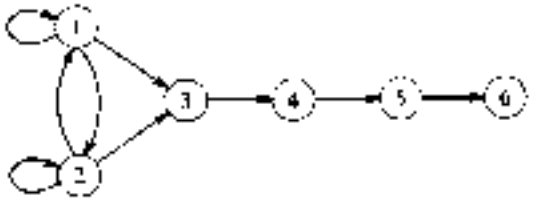


Fig. 3.43. Grafo orientado para a função *pos_seguinte*.

```
    marque j em T
    para cada símbolo de entrada a faça início
        seja P o conjunto de posições de estado em
            posicoes[j] que, para algum p em P, tal que
            estado[p] = q e simb[p] = a
        se P não estiver vazio e não estiver em Estados[T]
            então
            acrescente P como um estado em Estados[T]
            Dtran[i, a] = i
    fim
fim
```

Fig. 3.44 Construção de D_{tran} em 3.1

do de partição, de D_{tran} e de F, para obter os estados e as transições do AFD. Os estados são os elementos de Π_{final} que contêm pelo menos um estado de F. Seja s_0 o estado de Π_{final} que contém q_0, o estado inicial do AFN. Então s_0 é o estado inicial do AFD. Finalmente, as transições do AFD são definidas da seguinte maneira: para cada grupo G de Π_{final} e para cada símbolo de entrada a, examinamos $D_{tran}[q, a]$ para um estado q em G. Se $D_{tran}[q, a] = r$, e H é o grupo de Π_{final} que contém r, então o AFD tem uma transição do estado G para o estado H com símbolo de entrada a. Se $D_{tran}[q, a]$ não for definido para nenhum q em G, então o AFD não tem transição do estado G com símbolo de entrada a.

Exemplo 3.23. Vamos construir um AFD para a expressão regular `(a|b)*abb`. A árvore sintática para essa expressão foi construída na Fig. 3.30(a). Anotações de posições foram feitas em seguida à folha. As tabelas *posprim*, *posult* e *Seguidos* são mostradas na Fig. 3.31. A tabela *posicoes* aparece na Fig. 3.42.

A partir da Fig. 3.41, *posprim*[raiz] = {1, 2, 3}. Seja A este conjunto e consideremos o símbolo de entrada a. As posições em A são 1 e 3, e escrevemos $Ta = \{1, 3\}$. Como $posicoes[1] = posicoes[3] = a$, Ta é o conjunto de posições {1, 3}. Como este conjunto não é um dos estados em $Estados$, ainda, o acrescentamos a $Estados$. Para a entrada b, $Tb = \{2, 3\}$, e também acrescentamos Tb a $Estados$. Assim, $Estados = \{A, Ta, Tb\}$. Como A é o primeiro elemento de $Estados$ a ser marcado, consideramos A em seguida. Temos $Dtran[A, a] = Ta$ e $Dtran[A, b] = Tb$.

Quando consideramos o estado Ta, vemos que as posições em A somente 2 estão associadas com a entrada a. Seja Ua o conjunto $posicoes[2] = \{1, 2, 3\}$. Como este conjunto não está em $Estados$, ainda, acrescentamos Ua a $Estados$. Temos também $Dtran[Ta, a] = Ua$ e $Dtran[Ta, b] = Tb$.

Continuamos desta forma até $B = \{1, 2, 3, 4\}$. Os estados e transições que finalmente obtivemos são os mostrados na Fig. 3.43(b).

Minimizando o Número de Estados de um AFD

Um importante resultado teórico é que cada conjunto regular é reconhecido por um AFD mínimo que é único, a menos de nomes dos estados. Nesta seção, mostramos como construir este AFD mínimo pela redução do número de estados de um dado AFD sem alterar a linguagem reconhecida. Vamos supor que temos um AFD D com um conjunto de estados S, um alfabeto de símbolos de entrada Σ. Assumimos que cada estado possui uma transição em cada símbolo de entrada. Se isso não for o caso, introduzimos um novo "estado-morto" d, com transições de d para d em todas as entradas, e também uma transição de estado s para d para a entrada se não existir outra opção válida.

Dizemos que a cadeia x distingue o estado s do estado t se, quando começamos o AFD D pelo estado s e o alimentamos com a cadeia x, seguida pela entrada fim, dizemos que o estado de aceitação será atingido; mas se começamos pelo estado t e o alimentamos com a entrada x fim, dizemos que o estado de não aceitação será atingido, ou vice-versa. Por exemplo, a cadeia vazia distingue qualquer estado de aceitação de qualquer estado de não aceitação no AFD da Fig. 3.29. Os estados Ta e B são distinguidos pela entrada b, uma vez que Ta vai para o estado de não aceitação E, e o estado B vai para o estado de aceitação B, com a entrada b.

Nosso algoritmo para minimizar o número de estados de um AFD trabalha encontrando todos os grupos de estados que podem ser distinguidos por alguma cadeia de entrada. Os grupos de estados que não podem ser distinguidos são, então, mesclados em um único estado. Ou seja, mantemos um único estado para cada grupo, e todas as transições para estados do grupo são redirecionadas para o estado do grupo. Também mantemos apenas uma cópia de cada transição que deixa um estado do grupo. Se esses estados forem todos distinguidos, o AFD mínimo será o mesmo que o AFD original, a menos de nomes de estados.

Inicialmente, a partição consiste em dois grupos: os estados de aceitação e os de não aceitação. O básico do método é o seguinte: pegue algum estado s e algum símbolo de entrada a. Examine as transições para os estados t e u tais que s possui transições em a para t e u. Se t e u estiverem em grupos diferentes, então, para distinguir entre s e u, qualquer cadeia que comece com a distingue s de t. Portanto, os estados em s devem ser divididos de modo que s e u estejam em diferentes grupos da partição corrente. Supondo que todos os outros estados em t e u sejam divididos de modo que estejam em grupos separados, a é a única entrada que distingue s de t. Portanto, s deve ser dividido de acordo com a. Note que t e u são distinguidos por alguma cadeia de tamanho 2 ou mais, que s vai distinguir pela entrada a.

Repetimos o processo de examinar os grupos da partição em busca de grupos a serem divididos até que não seja mais possível encontrar grupos que precisem ser subdivididos. Quando isso acontece, temos a partição final. Escolhemos um estado em cada grupo para representar o grupo. Todas as transições a partir de um estado s são transições a partir do representante de s. Se t é um estado cujo grupo representante é u, então há uma transição do representante de s para o representante de u com o mesmo símbolo de entrada que a transição de s para t. Os estados de aceitação são representantes de grupos de aceitação, e o estado inicial é o representante do grupo que contém o estado inicial original. Se r é um representante de estado, e uma transição de r para q ocorre sob a entrada a, então q é o representante do grupo que contém $D_{tran}[r, a]$. Finalmente, se r é um representante de estado, então $D_{tran}[r, a]$ é definido se e somente se $D_{tran}[s, a]$ é definido para algum estado s no grupo de r.

Algoritmo 3.6. Minimizando o número de estados de um AFD.

Entrada. Um AFD D com conjunto de estados S, conjunto de entradas Σ, transições definidas por D_{tran}, estado de partida s_0 e conjunto de estados de aceitação F.

Saída. Um AFD D' que aceita a mesma linguagem que D e possui o menor número de estados possível.

Método.

1. Construa uma partição Π final Π de grupos de estados, sendo os grupos os estados de aceitação, F, e os de não aceitação, $S-F$.
2. Aplique o procedimento da Fig. 3.45. Ele cria de Π uma nova partição Π_{novo}.
3. Se $\Pi_{novo} = \Pi$, faça $\Pi_{final} = \Pi$ e continue com o passo 4. Em caso contrário, repita o passo 2 com $\Pi = \Pi_{novo}$.
4. Escolha um estado em cada grupo da partição Π_{final} como o representante desse grupo. Os representantes serão os estados do AFD D'.

```
para cada grupo G de Π faça início
    particione G em subgrupos tais que dois estados s e
        t de G estejam no mesmo subgrupo se e somente se, para
        todos os símbolos de entrada a, os estados s e t
        possuem transições com a para estados do mesmo
        grupo de Π
    /* no pior caso, um estado estará num subgrupo por si mesmo */
    substitua G em Π_novo pelo conjunto de todos os
        subgrupos formados
fim
```

Fig. 3.45 Construção de Π_{novo}

q_i, e k é M. Seja s um estado representante e suponhamos que a entrada a faz uma transição em M de s para t. Seja G o representante do grupo de G em que t se encontra. Então, M tem uma transição de s para t em a. Seja r o estado em P_i que representa o grupo G. Então, a função δ para o estado de partida s e entrada a é r. Os representantes de estados de aceitação de M são representantes de estados em F. Note-se que cada grupo em Π_{new} ou é constituído somente de estados em F ou não possui estados em F.

5. Se M possui um estado morto d, o estado s que lhe corresponde em G não possui transições para si mesmo em todos os símbolos de entrada. Como d de M é também um estado que não possui estados aceitadores para d de outros estados se tornam inalcançáveis. ■

Exemplo 3.24. A primeira reconstrução do AFD representado na Fig. 3.29. A partição inicial Π consiste em dois grupos: $G1$, o grupo dos estados de aceitação $\{A, B, D\}$, os estados de transição para aceitação. Para construir Π_{new}, o laço mais externo da Fig. 3.45 permite considerar $G1$. Como este grupo consiste de mais de um estado, cria-se $copy$ e cada um de $G1$ é colocado em Π_{new}. O algoritmo então considera o grupo $G2 = \{C\}$. A entrada a é a única dessas entradas possíveis. A função para A, B e D é, respectivamente, todos esses estados poderiam permanecer num mesmo grupo, caso fosse o único grupo a ser considerado. Verifica-se A entrando em $G1$ e C não para nenhum dos grupos. $G2 = \{C\}$ em Π, enquanto D vai para F, um membro de outro grupo. Então, em Π_{new}, o grupo $G2 = \{C\}$ precisa ser retirado em dois novos grupos, $G2 = \{C\}$ e $G3 = \{D\}$, e Π_{new} consiste em $\{G1, G2, G3\}$.

Na passagem seguinte do algoritmo da Fig. 3.45, de interesse somente a entrada a, mais $G1$ precisa ser repartido em dois novos grupos $G4 = \{A, B\}$ e $G5 = \{D\}$. Entrada b: A e C vão, cada um, a um estado em $G2$, enquanto B possui uma transição para D, um membro de um grupo de uma partição diferente da de C. O próximo novo Π é $\{G4, G2, G3, G5\}$.

Na passagem seguinte do algoritmo da Fig. 3.45 não podemos dividir mais nenhum grupo constituído de um único estado. A única possibilidade é a de subdividir $G4 = \{A, B\}$. Para a entrada a, um estado A e a entrada a vão para o mesmo estado $G4$; a entrada a também vai para o mesmo estado $G4$ a partir de B. Então, após essa passagem, $\Pi_{new} = \{G4, G2, G3, G5\} = \{\{A, B\}, \{C\}, \{D\}, \{E\}\}$.

Se escolhermos A como o representante do grupo $G4 = \{A, B\}$ e os mesmos B, D e E para os grupos com um único estado, a tabela de transições é mostrada na Fig. 3.46. O estado A é o estado de partida e os estados B e D são os estados de aceitação.

Por exemplo, no autômato reduzido, o estado A possui uma transição para o estado B com entrada b, dado que A representa o grupo de $G4$ e existe uma transição de A para C com entrada b no autômato original. Uma mudança similar teve lugar na interseção de A e entrada b, nesse exemplo — todas as outras transições foram copiadas a partir da Fig. 3.29. Não existe estado morto na Fig. 3.46 e todos os estados vão a ser acessados a partir do estado de partida A. ■

Estado	Símbolo de Entrada	
	a	b
A	B	A
B	B	D
D	B	E
E	B	A

Fig. 3.46 Tabela de transições para o AFD reduzido.

Minimização de Estados em Analisadores Léxicos

Para aplicar o procedimento de minimização para o AFD construído na Seção 3.7, precisamos somente o Algoritmo 3.5 e uma partição inicial que coloque em diferentes grupos todos os estados que individualmente indicam tokens diferentes.

Exemplo 3.25. No caso do AFD da Fig. 3.37, a partição inicial agrupa $\{0, 1, 3, 7, 8\}$ como S, uma vez que nenhum deles da indicação de reconhecer um token e $\{2, 4, 5, 6\}$ também serão agrupados, pois indicam o token a^*b^+. Os outros estados estarão em grupos de si mesmos. Então descobrimos imediatamente que $\{0, 1, 3, 7, 8\}$ e $\{2, 4, 5, 6\}$ pertencem a grupos diferentes, pois a vai para grupos existentes à entrada a. Igualmente, 2 e 5 não pertencem ao mesmo grupo pois 5 vai para um grupo à entrada b. Consequentemente, o AFD da Fig. 3.37 é o autômato de estado mínimo que realiza o trabalho. ■

Métodos de Compressão de Tabelas

Como indicamos, existem várias formas de se implementar a função de transição de um autômato. O processo de análise léxica ocupa uma parcela razoável do tempo de um compilador, uma vez que o GC precisa executar uma pesquisa para cada um dos caracteres de cada uma entrada. Consequentemente, o analisador léxico deve ser minimizado no número de operações que realiza por caractere de entrada. Se um AFD é usado para auxiliar a implementação de um analisador léxico, a forma de representação de escolha é a função de transição. A maneira padrão de representá-la é através de uma matriz indexada por estados e caracteres, pois oferece o acesso mais rápido. Pode ocupar muito espaço, dependendo do número de estados e de caracteres. Um esquema mais complexo pode ser mais lento, e não necessário. Isto ligado para otimizar as transições para fora de cada estado com uma transição "default" ao final da lista. A transição que ocorre mais frequentemente é a que se toma default.

Existe uma implementação mais sutil que é muito mais econômica em espaço e ainda razoavelmente eficiente. Ela utiliza duas estruturas de listas. Aqui usamos uma estrutura de dados que consiste em quatro ar-

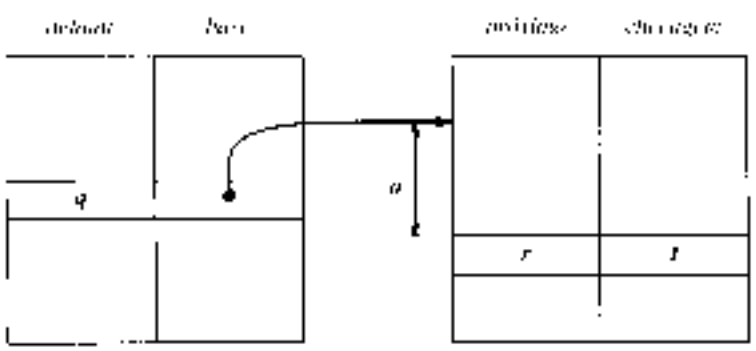


Fig. 3.47 Estrutura de dados para representar tabelas de transições

[illegible] indexação pelos números de estado, como ilustrado pela Fig. 3.27.[7] O array *base* é usado para determinar a localização-base de cada estado armazenada nos arrays *proximo* e *checagem*. O array *default* é usado para determinar uma localização-base alternativa, caso a localização-base corrente seja inválida.

Para computar *proximo*[*estado*], o próximo estado para a transição pelo estado *estado* e símbolo de entrada *a*, consultamos primeiro a parte do array *proximo* onde *checagem* tem particular o número das suas entradas para o estado *estado*, isto é, a localização *l* = *base*[*estado*] + *a*, onde *a* é tratado como inteiro. Tomamos *proximo*[*l*] como o próximo estado desde que a entrada *a* seja de fato esperada, isto é, *checagem*[*l*] = *estado*. Se *checagem*[*l*] ≠ *estado*, determinamos *q* = *default*[*estado*] e repetimos todo o procedimento recursivamente, usando *q* em lugar de *estado*. O procedimento é descrito a seguir:

```
procedimento proximo_estado(estado, a):
    se checagem[base[estado] + a] = estado então
        retornar proximo[base[estado] + a]
    senão
        retornar proximo_estado(default[estado], a)
```

O uso principal da estrutura da Fig. 3.27 é tornar os arrays *proximo* e *checagem* pequenos, aproveitando as vantagens de que muitas linhas diferentes de estados, isto é, estados para diferentes partes do autômato, podem ser combinadas, contanto que não interfiram umas com as outras. Por exemplo, o estado 0 depende principalmente do fato de que a entrada é o início da palavra-chave **then**, e também como um prefixo de um identificador. Ao caractere de entrada *t* pois usamos um estado especial que se lembra de que já examinamos *t*, mas que em outro caso corresponde ao estado 0. Em seguida, fazemos *base*[0] = 0 e *proximo*[*base*[0] + *t*] = 1. Também fazemos *checagem*[*base*[0] + *t*] = 0 para garantir que o estado 1 só seja consultado se a entrada anterior for *t*.

Contanto que evitemos explorar a escolha dos valores de *base*, de tal forma que ocorram colisões de *proximo* e *checagem* permanentes, nossa experiência mostra que esta estratégia simples de se estabelecer *base* para o menor número tal que as entradas especiais possam ser preenchidas sem conflito com as entradas já existentes é razoavelmente boa e se utiliza muito espaço além do mínimo possível.

Poderemos explorar a organização de uma tabela por estados se o AFD possuir a propriedade de que os lados incidentes para cada estado levam todos o mesmo rótulo *a*. Para implementar esse esquema, fazemos *checagem*[*l*] = *a* se substituirmos o teste na linha 2 do procedimento *proximo_estado* por

```
se checagem[proximo[base[estado] + a]] = estado então
```

EXERCÍCIOS

3.1 Qual é o alfabeto de entrada de cada uma das seguintes linguagens?
a) Pascal
b) C
c) Fortran 77
d) Ada
e) Lisp

3.2 Quais são as convenções relacionadas ao uso de espaços em cada uma das linguagens do Exercício 3.1?

3.3 Identifique os lexemes que consideramos fazer nos programas seguintes. Dê valores de atributo razoáveis para os tokens.

a) Pascal

[illegible]

b) C

[illegible]

c) Fortran 77

[illegible]

3.4 Escreva um programa para a função [illegible] da Seção 3.4 usando o esquema de bufferização com sentinelas descrito na Seção 3.2.

3.5 Numa cadeia de comprimento *n*, quantos dos seguintes elementos existem?
a) prefixos
b) sufixos
c) subcadeias
d) prefixos próprios
e) subsequências

***3.6** Descreva as linguagens denotadas pelas seguintes expressões regulares.
a) [illegible]
b) [illegible]
c) [illegible]
d) [illegible]
e) [illegible]

***3.7** Escreva definições regulares para as seguintes linguagens:
a) Todas as cadeias de letras que contenham as cinco vogais ordenadas.
b) Todas as cadeias de letras nas quais as letras estão em ordem lexicográfica ascendente.
c) Comentários consistindo de uma cadeia envolvida por /* e */ sem */* intervenientes a menos que ele apareça dentro de aspas.
***d)** Todas as cadeias de dígitos sem nenhum dígito repetido.
e) Todas as cadeias de dígitos com no máximo um dígito repetido.
f) Todas as cadeias de zeros e uns com um número par de zeros e ímpar de uns.
g) O conjunto de movimentos de cavalo em xadrez (ver como *PAR* no *CD*).
h) Todas as cadeias de zeros e uns que não contenham a subcadeia 011.
i) Todas as subcadeias de zeros e uns que não contenham a subsequência 011.

[7] Supomos que *proximo* e *checagem* têm o mesmo número de componentes, mesmo que não apareçam números consecutivos de estados. Na prática, os estados seriam agrupados, e só os grupos de estados (não os estados individuais) teriam entradas em *checagem* e *proximo*.

3.8 Especifique a forma lexical das constantes numéricas dadas nas alíneas do Exercício 3.1.

3.9 Especifique a forma lexical dos tokens delimitados por palavras chave e os seus lexemas do Exercício 3.1.

3.10 As combinações de expressões regulares permitidas por Lex estão listadas na Fig. 3.48 em ordem decrescente de precedência. Nessa tabela, r e s denotam duas expressões regulares quaisquer e n um inteiro positivo. Para cada item da lista, explique o significado da expressão regular e dê exemplos de cadeias que ela aceita.

a) O significado especial dos símbolos do operador

$$\backslash^{*} \; ? \; \$ \; [\;] \; ^{\wedge} \; - \; (\;) \; \{ \; \}$$

precisa ser desligado se o símbolo do operador deve ser usado como um caractere de reconhecimento. Desse modo, entre todos os operadores, só o caractere de barra invertida (\) precisa ser duplicado. Uma expressão "\n" reconhece uma cadeia de barra invertida seguida de n. Por exemplo, "*" reconhece a cadeia *. Podemos também reconhecer essa cadeia através de "[*]". Note que um asterisco com um operador de fecho de Kleene, como "*", é uma expressão regular que reconhece a cadeia "".

b) [illegible] Lex, [illegible] reconhece [illegible] qualquer caractere, exceto o de nova linha. Uma classe de caracteres [illegible] reconhece qualquer caractere que não seja um dos caracteres [illegible]. Mostre que [illegible] reconhece qualquer caractere que não seja uma letra maiúscula, minúscula ou um dígito. Mostre que, para cada definição regular com classes de caracteres complementadas, existe uma expressão regular equivalente sem os complementos.

c) A expressão regular r{m,n} reconhece de m a n ocorrências de r. Por exemplo, [illegible] reconhece um número de um a três dígitos. Mostre que, para uma expressão regular contendo operadores repetitivos, existe uma expressão regular equivalente sem os mesmos.

d) O operador ^ reconhece o início de uma linha. Este é o mesmo que o padrão que reconhece a classe de caracteres contendo apenas o caractere de nova linha. Explique por que o operador ^ só é significativo no início de uma expressão regular. Ilustre seu argumento mostrando como reescrever qualquer expressão regular que use ^ em outra que não o use. Sugestão: considere uma regra que reconheça a expressão regular seguida pelo operador ^.

3.11 Escreva um programa Lex que copie um arquivo, substituindo cada sequência não nula de espaços em branco por um único espaço.

3.12 Escreva um programa Lex que copie um programa Fortran, substituindo cada ocorrência de [illegible] por [illegible].

3.13 Use a especificação para palavras chave e identificadores do Problema 2.7 e a parte do Exercício 3.9 para identificar os tokens dos seguintes enunciados:

```
[illegible]
[illegible]
[illegible]
[illegible]
...
```

Você pode escrever uma especificação para palavras chave e identificadores em Lex?

3.14 No sistema UNIX, o comando de *shell* [illegible] usa os operadores da Fig. 3.48 (e, supõe-se, suas formas de arquivos) como descritores de conjuntos de nomes de arquivos. Por exemplo, a especificação de nome de arquivo [illegible] reconhece todos os nomes de arquivo terminados em [illegible] reconhece todos os nomes que são da forma [illegible] (qualquer caractere, seguido de x, seguido de qualquer outro caractere). Nomes de arquivos podem ser abreviados como em [illegible]. Mostre como essas expressões de nomes de arquivo [illegible] podem ser representadas por expressões regulares.

3.15 Modifique o Algoritmo 3.4, a fim de que encontre o mais longo prefixo da entrada que seja aceito pelo AFD.

3.16 Construa um autômato finito determinístico para as seguintes expressões regulares usando o Algoritmo 3.5. Mostre a sequência de movimentos feita por cada um no processamento da cadeia de entrada *ababb*.

a) [illegible]

b) [illegible]

c) [illegible]

d) [illegible]

3.17 Converta os AFNs do Exercício 3.16 em AFDs usando o Algoritmo 3.2. Mostre as sequências de movimentos feitas por cada um no processamento da cadeia de entrada *ababb*.

3.18 Construa AFDs para as expressões regulares do Exercício 3.16 usando o Algoritmo 3.5. Compare o tamanho dos AFDs com aqueles construídos no Exercício 3.17.

3.19 Construa um autômato finito determinístico para os dígitos de transições dos *gotos* da Fig. 3.47.

3.20 Amplie a tabela da Fig. 3.48 de modo a incluir os operadores de expressões regulares [illegible].

3.21 Minimize o número de estados dos AFDs do Exercício 3.16 usando o Algoritmo 3.6.

3.22 Podemos provar que duas expressões regulares são equivalentes mostrando que seus AFDs mínimos são os mesmos, a menos dos nomes de estados. Usando esse teorema, provar que as seguintes expressões regulares são equivalentes:

a) [illegible]

b) [illegible]

c) [illegible]

Expressão	Elementos Reconhecidos	Exemplo
[illegible]	[illegible]	[illegible]
[illegible]	qualquer caractere, menos o operador	[illegible]
[illegible]	caractere literalmente	[illegible]
[illegible]	cadeia literalmente	[illegible]
[illegible]	qualquer caractere após a barra de folha	[illegible]
[illegible]	início de linha	[illegible]
[illegible]	fim de linha	[illegible]
[illegible]	qualquer caractere em [illegible]	[illegible]
[illegible]	qualquer caractere, exceto os de [illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]

Fig. 3.48. Expressões regulares de Lex.

Expressão	Elementos Reconhecidos	Exemplo
[illegible]	cadeia literalmente	[illegible]
[illegible]	caractere literalmente	[illegible]
[illegible]	qualquer cadeia	[illegible]
[illegible]	qualquer caractere	[illegible]
[illegible]	qualquer caractere em [illegible]	[illegible]

Fig. 3.49. Expressões comuns de arquivos em programas [illegible].

3.23 Construa um AFD mínimo para cada uma das seguintes expressões regulares.
a) $(a|b)^*(aa|bb)(a|b)^*$
b) $(a|b)^*(aa|bb)(a|b)^*(aa|bb)(a|b)^*$
c) $(a|b)^*aa(a|b)^*bb(a|b)^*|(a|b)^*bb(a|b)^*aa(a|b)^*$
*d) Prove que qualquer autômato finito determinístico para a expressão regular $(a|b)^*aa(a|b)^*bb(a|b)^*|(a|b)^*bb(a|b)^*aa(a|b)^*$, onde existam $n > 0$ e $m > 0$ tais que $n \neq m$, precisa ter pelo menos 2^k estados.

3.24 Construa uma representação da Fig. 3.47 para a tabela de transições do Exercício 3.19. Observe que os estados *desconhecido* e nomes são dois segmentos recebidos para construir o nome *variável* e computar as quantidades desses segmentos.
a) Começando pelos estados mais densos (aqueles com o maior número de entradas diferentes de seus estados *default*), coloque primeiro as entradas para aqueles estados no vetor *proximo*.
b) Coloque os estados para os estados menos densos, procurando uma ordem randômica.

3.25 Uma variação do esquema de compressão de tabela da Seção 3.9 seria evitar o procedimento recursivo *proximo-estado* através de uma localização linear para o estado base *localização-default* de cada estado. Construa a representação da Fig. 3.47 para a tabela de transições do Exercício 3.19 usando esta técnica não recursiva. Compare as exigências de espaço com aquelas do Exercício 3.24.

3.26 Seja $b_1 b_2 \ldots b_n$ uma cadeia padrão, chamada de *palavra-chave*. Defina para uma palavra-chave um diagrama de transições com $n+1$ estados, no qual cada estado corresponde a um prefixo de palavra-chave. Para $1 \leq i \leq n$, existe uma transição do estado $i-1$ para o estado i rotulado b_i. Os números no canto final correspondem à cadeia casada após a palavra-chave correspondente. Assim para a palavra-chave *ababaca*:

Defina uma *função de falha* para cada estado do diagrama de transições, exceto para o estado de partida. Suponha que queremos estender o reconhecimento por transições da palavra-chave. Definimos, então, $f(n) = i$ se e somente se os i mais longos estados próprios de n que também são prefixos de uma palavra-chave. A função de falha f para o *trie* acima é:

s	1	2	3	4	5	6
$f(s)$	0	0	1	2	3	1

Por exemplo, os estados 3 e 1 representam os prefixos *aba* e *a* da palavra-chave *ababaca*, e $f(3) = 1$ pois *a* é o sufixo próprio mais longo de *aba* que é prefixo da palavra-chave.
a) Construa a função de falha para a palavra-chave *ababababa*.

```
/* computar a função de falha f para b_1 ... b_n */
f(1) := 0; f(0) := 0;
para i := 1 até n - 1 faça início
    enquanto i > 0 e b_{i+1} ≠ b_s faça s := f(s);
    se b_{i+1} = b_s então início f(i+1) := s + 1; fim;
    senão f(i+1) := 0
fim
```

Fig. 3.50. Algoritmo para computar a função de falha para o Exercício 3.26.

```
/* cadeia[1:m] é uma subcadeia de b_1 ... b_n */
s := 0;
para i := 1 até m faça início
    enquanto s > 0 e cadeia[i] ≠ b_{s+1} faça s := f(s);
    se cadeia[i] = b_{s+1} então início s := s + 1;
    se s = n então retornar "sim" fim;
fim;
retornar "não"
```

Fig. 3.51. Algoritmo KMP.

*b) Sejam os estados do AFD $0, 1, \ldots, n$ com 0 o único estado de parada. Mostre que o algoritmo da Fig. 3.50 computa corretamente a função de falha.
*c) Mostre que na execução global do algoritmo da Fig. 3.50 o comando $s := f(s)$ no laço interno nunca é executado mais do que n vezes.
*d) Mostre que o algoritmo executa no máximo $2n$ vezes.

3.27 O Algoritmo KMP da Fig. 3.51 usa a função de falha construída no Exercício 3.26 para determinar se uma palavra chave $b_1 b_2 \ldots b_n$ é uma subpalavra de uma cadeia padrão $a_1 a_2 \ldots a_m$. Os estados para o AFD $b_1 b_2 \ldots b_n$ são numerados de 0 a n, como no Exercício 3.26(b).
a) Aplique o Algoritmo KMP para determinar se *ababaca* é uma subpalavra de *abacadabrabacabada*.
*b) Prove que o Algoritmo KMP retorna "sim" se e somente se $b_1 \ldots b_n$ é uma subpalavra de $a_1 \ldots a_m$.
*c) Mostre que o Algoritmo KMP roda num tempo $O(m+n)$.
*d) Dada uma palavra-chave $b_1 \ldots b_n$, mostre que a função de falha pode ser usada para construir um tempo $O(n)$ para AFD com $n+1$ estados para a expressão regular $\Sigma^*(b_1|b_2|\ldots|b_n)$, onde Σ denota os pontos de qualquer caractere de entrada.

++3.28 Defina que um *período* de uma cadeia s como sendo um inteiro p tal que s possa ser expresso como $(xy)^i x$ para algum $i \geq 0$, onde $xy = s$ e x ou y (ou ambos) é a cadeia vazia. Por exemplo, 2 e 4 são períodos da cadeia *ababa*.
a) Mostre que q é um período de uma cadeia s se e somente se $s = pq$ para algumas cadeias p e q de comprimento q.
b) Mostre que se p e q não são períodos de uma cadeia s e $p + q - \text{mdc}(p, q)$ é menor ou igual ao comprimento de s, onde mdc denota o maior divisor comum de p e q.
c) Seja p o menor período de uma cadeia s. Mostre que a função de falha fornece o período p de s em $f(n) = p - n$.

+3.29 Seja o *vetor prefixo* associado a uma cadeia s o vetor cujos prefixos de s são $s_1, s_2, \ldots, s_n$ para algum $n \geq 1$. Por exemplo, os vetores prefixos correspondentes às palavras *a*, *aa*, *aaa* e *ab* são *a*, *aa*, *aaa* e *ab*. Construa um algoritmo que encontra o vetor prefixo de s em tempo $O(n)$. Sugestão: Use a função de falha do Exercício 3.26.

3.30 Uma *cadeia de Fibonacci* é definida como se segue:

$s_1 = b$
$s_2 = a$
$s_k = s_{k-1} s_{k-2}$, para $k > 2$.

Por exemplo, $s_3 = ab$, $s_4 = aba$ e $s_5 = abaab$.
a) Qual é o comprimento de s_k?
*b) Qual é o menor período de s_k?
c) Construa a função de falha para s_k.
*d) Usando a função, mostre que a função de falha para s_k pode ser expressa por $f(j) = j - s_{k-1}$, onde j é tal que $s_{k-1} = 1 \ldots s_{k-1}$, para $1 < j \leq |s_k|$.
e) Aplique o Algoritmo KMP para determinar se s_5 é uma subcadeia da cadeia *abaababaabaab*.
f) Construa um AFD para a expressão regular $(a|ab)^*$.

(p) No algoritmo KMP, qual é o número máximo de atribuições consecutivas da função de falha executadas ao se determinar se x é uma subcadeia da cadeia-alvo y?

3.31 Podemos estender os conceitos de *prefixo* e de função de falha, do Exercício 3.29, de uma palavra-chave para um conjunto de palavras-chave, como se segue. Cada estado no trie corresponde a um prefixo de uma ou mais palavras-chave. O símbolo de parada corresponde a cada vértice interior que corresponde a uma palavra-chave completa e um estado final. Estados finais podem ser tratados como estados finais durante a computação da função de falha. O diagrama de transições para o conjunto de palavras-chave {he, she, his, hers} é mostrado na Fig. 3.52.

Em vez de definirmos uma *função de transição* que mapeia pares estado-símbolo para estados tal que $g(s, b) = s'$ se o estado s corresponde a um prefixo $b_1b_2 \ldots b_j$ de alguma palavra-chave e s' corresponde ao prefixo $b_1b_2 \ldots b_jb$, $s = 0$ é o estado de partida, definimos $g(s, b) = fail$ para todos os símbolos de entrada b que não são rótulo de qualquer transição válida de s. Fazemos também $g(0, b) = fail$ para qualquer transição válida do estado de partida. Note-se que não existem transições *fail* para o estado de partida.

Suponhamos que os estados 0 a 9 representam os prefixos e, she, h, his, he, hers, s, sh, her, hi, respectivamente. Definimos então $f(s) = s'$ se o prefixo correspondente ao estado s' também seja um sufixo de algum prefixo de alguma palavra-chave. A função de falha f para o diagrama de transições abaixo é:

s	1	2	3	4	5	6	7	8	9
$f(s)$	0	0	0	1	2	0	3	0	3

Por exemplo, os estados 4 e 5 representam os prefixos $s = \text{hi}$ e $t = \text{his}$. Temos $f(5) = 3$ porque s é o prefixo próprio mais longo de t que é também sufixo de algum prefixo de alguma palavra-chave. A função de falha f pode ser computada para os estados de profundidade d recursivamente usando o algoritmo da Fig. 3.53. A profundidade de um estado é a sua distância a partir do estado de partida.

Note-se que, como $g(0, x) = fail$ para qualquer caractere x que não corresponda ao início de uma palavra-chave, após fazer $s = g(s, b)$ igual a $fail$, se $g(s, x) \neq fail$ temos também $g(s, x)$ um estado final, se quisermos saber se x contém uma palavra-chave.

(a) Construir a função de falha para o conjunto de palavras-chave {poe, poem, poet, poetry}.

(b) Mostre que o algoritmo da Fig. 3.53 computa corretamente a função de falha.

(c) Mostre que a função de falha pode ser computada em tempo proporcional à soma dos comprimentos das palavras-chave.

3.32 Seja g a função de transição e f a função de falha do Exercício 3.31 para um conjunto de palavras-chave $K = \{y_1, y_2, \ldots, y_k\}$. O Algoritmo AC da Fig. 3.54 usa g e f para determinar se uma cadeia-alvo $x = x_1x_2 \ldots x_n$ contém uma subcadeia que seja uma palavra-chave. Os estados de estado de parada do diagrama de transições para K e F é o conjunto de estados finais.

(a) Aplique o Algoritmo AC à cadeia de entrada $x = \text{ushers}$ usando as funções de transição e de falha do Exercício 3.31.

(b) Prove que o Algoritmo AC retorna "sim" se e somente se alguma palavra-chave y for uma subcadeia de x.

(c) Mostre que o Algoritmo AC realiza, no pior caso, $2n$ transições ao processar uma cadeia de entrada de comprimento $n \geq 0$.

(d) Mostre que a partir de um diagrama de transições e da função de falha para um conjunto de palavras-chave

$$\{y_1, y_2, \ldots, y_k\} \text{ tais que } \sum_{i=1}^{k} |y_i| = l$$

o Algoritmo AC executa no máximo $2n - l$ estados, onde n pode ser considerado uma função linear para a expressão $2n - l$ (ou seja, $c_1n + c_2$ para algumas constantes $c_1 > 0$ e $c_2 \geq 0$).

(e) Modifique o Algoritmo AC para imprimir cada palavra-chave encontrada na cadeia-alvo.

3.33 Use o Algoritmo do Exercício 3.32 para construir um analisador léxico para as palavras-chave em Pascal.

3.34 Defina o *comprimento da subsequência comum mais longa* de duas cadeias x e y como a cadeia que seja uma subsequência de ambas x e y e que seja tão longa quanto qualquer subsequência comum. Por exemplo, $z = \text{aab}$ é a mais longa subsequência de $x = \text{aabb}$ e $y = \text{baab}$. Defina a *distância de edição* entre x e y como sendo o número mínimo de inserções e remoções requeridas para transformar x em y. Por exemplo, $d(x, y) = 3$ e $d(y, z) = 2$.

(a) Mostre que para quaisquer duas cadeias x e y, a distância de edição $d(x, y)$ é igual à soma dos comprimentos de x e y menos duas vezes o comprimento da subsequência comum mais longa de x e y.

(b) Escreva um algoritmo que tome duas cadeias x e y como entrada e produza a subsequência comum mais longa de x e y como saída.

3.35 Defina o *custo de edição* entre duas cadeias x e y, como sendo o menor custo de inserções, remoções e substituições que são requeridas para transformar x em y. Se $x = a_1a_2 \ldots a_m$ e $y = b_1b_2 \ldots b_n$, $d(x, y)$ pode ser computada por um algoritmo de programação dinâmica usando uma matriz de custos $d[0..m, 0..n]$, na qual $d[i, j]$ é o custo de edição entre $a_1a_2 \ldots a_i$ e $b_1b_2 \ldots b_j$. Assuma que o custo de inserção, remoção e substituição é 1 cada.

```
para cada estado s de profundidade 1 faça
    f(s) = 0
para cada profundidade d = 1 até ... faça
    para cada estado s de profundidade d e cada letra c tal
        que g(s, c) = s' faça início
        s = f(s)
        enquanto g(s, c) = fail faça s = f(s)
        f(s') = g(s, c)
fim
```

Fig. 3.53. Algoritmo para computar a função de falha para um conjunto de palavras-chave

```
/* a cadeia x contém uma palavra-chave como subcadeia se
   o estado final for atingido */
s = 0
para i = 1 até n faça início
    enquanto g(s, x_i) = fail faça s = f(s)
    s = g(s, x_i)
    se estado s for final então retornar "sim"
fim
retornar "não"
```

Fig. 3.54. Algoritmo AC

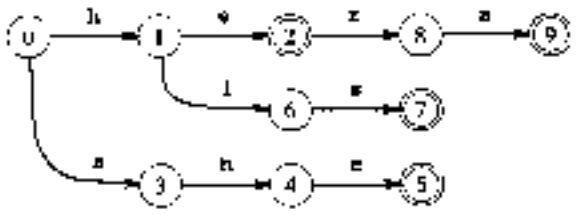


Fig. 3.52. Trie para as palavras-chave

```
para i := 0 até m faça d[i, 0] := i
para j := 1 até n faça d[0, j] := j
para i := 1 até m faça
    para j := 1 até n faça
        d[i, j] := mínimo de [d[i-1, j] + 1, d[i, j-1] + 1,
                              d[i-1, j-1] + custo(x_i, y_j)]
```

Fig. 3-35. Algoritmo para computar a distância mínima entre duas cadeias

c) (*Difícil*) O algoritmo da Fig. 3-35 pode ser usado para computar a função *custo*. A função *custo* é claramente o custo da substituição de um caractere x_i por y_j, isto é, 0 se $x_i = y_j$ e 1 em caso contrário.

a. Qual é a relação entre a medida de distância do Exercício 3-34 e a distância de edição?

b. Use o algoritmo da Fig. 3-35 para computar a distância de edição entre *abcdefgh* e *bcdefgha*.

c. Construa um algoritmo que, dadas duas cadeias x e y, encontre uma de menor comprimento de edição exponente para transformar x em y.

3.36 (*Difícil*) Escreva um algoritmo que tome como entrada uma cadeia x e uma expressão regular r e produza como saída uma cadeia em $L(r)$ (se houver) que seja a mais pequena quanto possível com relação à função de distância do Exercício 3-34.

EXERCÍCIOS DE PROGRAMAÇÃO

P3.1 Escreva um analisador léxico em Pascal, Ratfor ou outra linguagem de alto nível para os tokens descritos na Fig. 3-10.

P3.2 Escreva uma especificação sintática *à la* Pascal e, a partir dela, construa os diagramas de transições. Use os métodos para implementar uma rotina de análise léxica para Pascal numa linguagem como C ou Pascal.

P3.3 Complete o programa Lex na Fig. 3-18. Compare o tamanho e a velocidade do analisador léxico resultante produzido por Lex e com o programa escrito no Exercício P3.1.

P3.4 Escreva uma especificação Lex para os tokens em Pascal e use o compilador Lex para construir um analisador léxico para Pascal.

P3.5 Escreva um programa que tome como entrada uma expressão regular e o nome de um arquivo e produza como saída todas as linhas do arquivo que contenham uma subcadeia denotada pela expressão regular.

P3.6 Adicione um esquema de recuperação de erros ao programa Lex na Fig. 3-18 a fim de capacitá-lo a usar a função *pesquisa* para localizar palavras ortograficamente próximas.

P3.7 Programe um analisador léxico a partir do AFD construído no Exercício 3-18 e compare este analisador léxico com aquele construído nos Exercícios P3.1 e P3.3.

P3.8 Construa um compilador que produza um analisador léxico a partir da descrição sob a forma de expressões regulares de um conjunto de tokens.

NOTAS BIBLIOGRÁFICAS

A descrição formal dos aspectos léxicos de uma linguagem surgiu principalmente de dificuldades vividas pelo ambiente no qual a linguagem foi criada. Quando Fortran foi projetado em 1954, os cartões perfurados eram um meio comum de entrada. Os espaços não foram ignorados em Fortran porque os digitadores que preparavam os cartões a partir de anotações manuscritas frequentemente estimavam erroneamente os espaços (Backus [1981]). A separação, em Algol 58, da representação de *hardware* e da linguagem de referência é um compromisso criado durante um encontro de cúpula de projeto de linguagem (Naur [1981]). Uma vez que

Quando o custo de um ponto indicando um valor de erro é o custo de um ponto decimal (Wagstaff [1981]).

Knuth [1973] apresenta técnicas adicionais para bufferização de entrada. Isshiki [1979] discute as dificuldades práticas do reconhecimento de *tokens* em Fortran 77.

As expressões regulares foram primeiro estudadas por Kleene [1956], que estava interessado na descrição dos eventos que poderiam ser representados pelo modelo de computação de uma determinada rede neural de McCulloch e Pitts [1943]. A minimização dos autômatos finitos foi primeiramente estudada por Huffman [1954] e Moore [1956]. A equivalência entre autômatos determinísticos e não determinísticos bem como outros aspectos desses reconhecedores foi mostrada por Rabin e Scott [1959]. McNaughton e Yamada [1960] descreveram um algoritmo para construir um AFD diretamente a partir de uma expressão regular. Muitos resultados da teoria dos autômatos podem ser encontrados em Hopcroft e Ullman [1979].

Foi rapidamente compreendido que as ferramentas para construir analisadores léxicos a partir de expressões regulares seriam úteis na implementação de compiladores. Johnson *et al.* [1968] discutem um sistema precursor (Lex) a linguagem discutida neste capítulo e devida a Lesk [1975], e também usada para construir analisadores léxicos para muitos compiladores utilizados no sistema UNIX. Desperta-se de implementação compacta das tabelas de transições na Seção 3.9 se deve a S.C. Johnson que projetou o primeiro gerador de varredores Yacc (Johnson [1975]). Outros esquemas de compressão de tabelas são discutidos e avaliados em Dencker, Dürre e Heuft [1984].

O problema da implementação compactada das tabelas de transições foi teoricamente estudado por Tarjan e Yao [1979] e por Fredman, Komlós e Szemerédi [1984]. Cormack, Horspool e Kaiserswerth [1985] apresentam um algoritmo perfeito de hashing, pesquisa nesse tabela.

As expressões regulares estendidas foram estudadas em muitas aplicações além da compilação. Manuais editores de texto usam expressões regulares para pesquisas de contexto. Thompson [1968], por exemplo, descreve a construção de um AFN a partir de uma expressão regular. O algoritmo 3.3 foi usado no editor QED. O sistema UNIX possui dois programas de propósito geral de pesquisa em expressões regulares [illegible]; grep permite a utilização de operadores para agrupar subexpressões regulares e também uma forma limitada de referenciamento posterior (isto é, referência a uma parte anterior do elemento previamente localizado, como em Snobol [illegible] empregados Algoritmos 3.1 e 3.4 para pesquisar sequências sob a forma de expressões regulares. As expressões regulares estendidas [illegible] são similares àquelas do Lex e foram baseadas em *hoc* [illegible] [illegible] com uma extensão pregressa destinada a fim de pesquisar seus padrões sob a forma de expressões regulares, como descrito na Seção 3.7. [illegible] busca pelos padrões que consistem em conjuntos de palavras chave usando o algoritmo 3 em Aho e Corasick [1975], o qual é discutido nos Exercícios 3.31 e 3.32. Aho [1980] discute o desempenho e a forma desses programas.

As expressões regulares têm sido amplamente usadas em sistemas de recuperação de textos, em linguagens de interrogação de bancos de dados e linguagens de processamento de arquivos, como AWK (Aho, Kernighan e Weinberger [1979]), Icon (Griswold [1976]) usam expressões regulares para descrever as imperfeições dos elementos impressos. Cherry [1982] descreve um programa de escrever automaticamente palavras-chave do Exercício 3.35 para pesquisar palavras mal escritas em manuscritos.

O algoritmo de reconhecimento de padrões para cadeias estendidas nos Exercícios 3.26 e 3.27 é de Knuth, Morris e Pratt [1977]. Esse artigo

---

A linguagem de descrição de hardware ISP é descrita em Bell e Newell [1971]. Uma descrição mais recente de uma linguagem de descrição de hardware é dada em Barbacci [1981]. Uma coleção de modelos de computadores escritos em uma linguagem de descrição de hardware é dada em Siewiorek, Bell e Newell [1982]. Uma linguagem de descrição de hardware é uma linguagem de programação que contém construções para descrever a estrutura e o comportamento de circuitos digitais. Uma linguagem de descrição de hardware pode ser usada para simular um circuito digital antes que ele seja construído. (N. do T.)

também contem uma boa discussão de buscas em cadeias. Outro algoritmo bastante poderoso para reconhecimento de cadeias foi descoberto por Boyer e Moore [1977], que mostraram que um reconhecimento de uma subcadeia pode normalmente ser determinado sem se ter que examinar todos os caracteres da cadeia alvo. O algoritmo também foi provado como um teste eficiente para possíveis ocorrências de padrões de cadeias [Hancock 1975].

A sequência subsequência comum mais longa discutida no Exercício 3.31 foi usada no projeto do programa de computação de sistema de arquivos do UNIX [Hunt e McIlroy 1976]. Um algoritmo eficiente e prático para o cômputo das subsequências comuns mais longas é descrito em Hunt e Szymanski [1977]. O algoritmo para computar as distâncias mínimas de edição do Exercício 3.35 está em Wagner e Fischer [1974]. Wagner [1974] contém uma solução para o Exercício 3.36. Sankoff e Kruskal [1983] contem uma discussão fascinante sobre o amplo leque de aplicações dos algoritmos de reconhecimento de distância mínima para o estudo de padrões em sequências gerenciais e problemas no processamento da linguagem falada.

CAPÍTULO 4

ANÁLISE SINTÁTICA

Cada linguagem de programação possui as regras que descrevem a estrutura sintática de programas bem-formados. Em Pascal, por exemplo, um programa é constituído por blocos, um bloco por comandos, um comando por expressões, uma expressão por fatores e assim por diante. A análise das construções de uma linguagem de programação pode ser descrita pela gramática livre de contexto ou pela notação BNF (Forma de Backus-Naur), introduzidas na seção 2.2. As gramáticas oferecem vantagens significativas tanto para os projetistas de linguagens quanto para os escritores de compiladores.

- Uma gramática bem-escrita para uma linguagem de programação fornece uma especificação sintática precisa e fácil de entender.
- Para certas classes de gramáticas, podemos construir automaticamente um analisador sintático que determina se um programa fonte está sintaticamente bem-estruturado. Uma boa especificação gramatical também pode revelar ambiguidades sintáticas bem como outras construções difíceis de se analisar e, portanto, indesejáveis. Algumas poderão, de outra forma, escapar inteiramente à fase de projeto inicial de uma linguagem e de seu compilador.
- Uma gramática cuidadosamente projetada implica uma estrutura de linguagem de programação útil à tradução correta de programas-fonte e, em seguida, à geração de código-objeto e também à detecção de erros. Existem ferramentas disponíveis para a conversão de descrições de tradução baseadas em gramáticas, em programas operativos.
- As linguagens evoluem ao longo de um certo período de tempo, adquirindo novas construções e realizando tarefas antigas de maneiras novas. Essas novas construções podem ser mais facilmente incluídas em uma implementação baseada numa descrição gramatical da linguagem.

O restante deste capítulo está devotado aos métodos de análise sintática que são apropriados para uso em compiladores. Apresentamos primeiramente os conceitos básicos, em seguida as técnicas adequadas à implementação manual e, finalmente, os algoritmos usados em ferramentas automatizadas. Como os programas podem conter erros sintáticos, estudamos os métodos de análise de forma que se recuperem dos erros que ocorrem mais comumente.

4.1 O PAPEL DO ANALISADOR SINTÁTICO

Em nosso modelo de compilador, o analisador sintático recebe uma cadeia de tokens provenientes do analisador léxico, como mostrado na Fig. 4.1, e verifica se a cadeia pode ser gerada pela gramática da linguagem-fonte. Esperamos que o analisador sintático relate qualquer erro de sintaxe de uma forma inteligível. Deve também ser recuperar dos erros que ocorrem mais comumente, a fim de poder continuar processando o resto do seu programa.

Existem três tipos gerais de analisadores sintáticos. Os métodos universais de análise sintática tais como o algoritmo de Cocke-Younger-Kasami e o de Earley, podem tratar qualquer gramática (ver as referências bibliográficas). Esses métodos, entretanto, são muito ineficientes para serem empregados em produção. Os métodos usados comumente nos compiladores são classificados como *top-down* ou *bottom-up*. Como indicado por seus nomes, os analisadores sintáticos *top-down* constroem árvores sintáticas a partir do topo (raiz) e do início para baixo, enquanto que os *bottom-up* começam pelas folhas e trabalham até alcançar a raiz. Em ambos os casos a entrada é varrida da esquerda para a direita, um símbolo de cada vez.

Os métodos de análise mais eficientes, tanto *top-down* quanto *bottom-up*, trabalham somente em determinadas subclasses de gramáticas, mas várias dessas subclasses, como as das gramáticas LL e LR, são suficientemente expressivas para descrever a maioria das construções sintáticas das linguagens de programação. Os analisadores implementados manualmente podem lidar frequentemente com gramáticas LL. Por exemplo, a abordagem da Seção 2.4 constrói analisadores

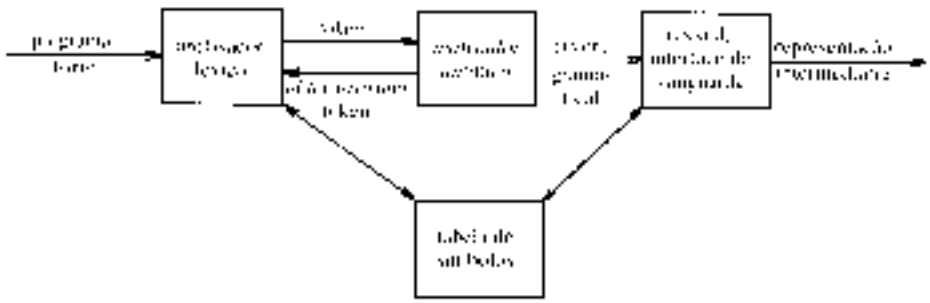


Fig. 4.1. Posição do analisador sintático em um modelo de compilador

1 Existem métodos mais eficientes para classes especiais de gramáticas. Por exemplo, temos métodos de análise de tempo linear para gramáticas LR. Veja a Seção 4.7.

sintáticos são gramáticas livres de contexto. Os detalhes das classes mais complexas de gramáticas livres de contexto são frequentemente construídos através de ferramentas automatizadas.

Neste capítulo, vamos estudar as técnicas usadas por tradutores e compiladores para representação de árvores sintáticas e para verificação de que um programa-fonte obedece às regras de sintaxe da linguagem de programação. Na prática, existem muitos detalhes menores de um tradutor que podem ser convenientemente tratados durante a análise sintática, tais como coletar informações sobre os vários tokens no programa-fonte, inserir informações em tabelas de símbolos e realizar a detecção de tipos e outras formas de análise semântica, e também gerar o código intermediário (ver Capítulo 2, Interações entre as fases de um tradutor" da Fig. 4.1). Vamos discutir esses atividades e os detalhes nos capítulos seguintes.

Nesta seção, descrevemos as funções de tratamento de erros sintáticos e as estratégias gerais para a sua implementação. Duas dessas estratégias chamadas modalidade desespero e correção em nível de frase, são discutidas mais adiante neste capítulo, porque normalmente estão associadas a métodos específicos de análise sintática. A implementação de uma boa estratégia tem o potencial de poupar muito esforço na depuração de um programa-fonte incorreto e de produzir um tradutor mais fácil de usar. Felizmente, muitas técnicas úteis podem ser implementadas com modificações relativamente pequenas em um analisador sintático.

Tratamento dos Erros de Sintaxe

Se um compilador tivesse que processar somente programas corretos, seu projeto e sua implementação seriam grandemente simplificados. Mas os programadores frequentemente escrevem programas incorretos, e um bom compilador deve auxiliar o programador na identificação e localização de erros. Programadores, que apesar de seus erros sempre lógicos, contam com uma pequena linguagem para produzir rapidamente um programa executável, vão considerar isso como uma característica importante dessa linguagem. Na verdade, os programadores experientes frequentemente preferem uma linguagem que rejeita programas incorretos a uma que aceita programas incorretos, mas depois falha durante a execução. Problemas que são capturados em tempo de compilação podem ser corrigidos mais rapidamente do que problemas que só se manifestam em tempo de execução. Além disso, a localização de erros feita pelo compilador pode muitas vezes apontar diretamente para a causa do problema, enquanto que um erro detectado em tempo de execução pode ter seus sintomas muito distantes de sua causa, tornando a depuração problemática. Finalmente, um compilador pode tentar reparar erros de modo que possa detectar mais erros em uma única execução. A aplicação bem-sucedida de qualquer uma dessas características aumenta drasticamente a carga de trabalho inicial do projetista de compiladores. No entanto, a adição de uma quantidade razoável de esforço à implementação de um compilador para que ele seja mais amigável pode ser uma grande vantagem.

Sabemos que os programas podem conter erros em muitos níveis diferentes. Por exemplo, os erros podem ser

- léxicos, tais como erros na grafia de um identificador, palavra-chave ou operador
- sintáticos, tais como uma expressão aritmética com parênteses não balanceados
- semânticos, tais como um operador aplicado a um operando incompatível
- lógicos, tais como uma chamada infinitamente recursiva

Frequentemente, uma parte da detecção e recuperação de erros em um compilador gira em torno da fase de análise sintática. Isso porque ou os erros são sintáticos por natureza ou são apresentados quando o fluxo de tokens proveniente do analisador léxico desobedece as regras gramaticais que definem a linguagem de programação. Outra razão está na precisão com os modernos métodos de análise sintática podem detectar muito rapidamente a presença de erros sintáticos indesejados. Detectar posteriormente a presença de erros semânticos em tipos em um compilador é uma tarefa muito mais difícil. Nesta seção, apresentaremos várias noções básicas para a recuperação de erros sintáticos. Suas implementações são discutidas em conjunto com os métodos de análise sintática deste capítulo.

O tratador de erros de um analisador sintático possui metas simples de serem estabelecidas:

- Deve relatar a presença de erros claramente e exatamente.
- Deve se recuperar de cada erro rapidamente o bastante para ser capaz de detectar erros subsequentes.
- Não deve retardar significativamente o processamento de programas corretos.

A realização eficiente dessas metas apresenta desafios difíceis.

Felizmente, as estratégias mais simples frequentemente exibem bom comportamento. Não existe um mecanismo de tratamento de erros relativamente claro e simples para todos os erros sintáticos. Com efeito, se muitos erros permanecem no programa-fonte, o tradutor pode ficar confuso com o contexto fornecido e pode ser incapaz de fazer muitas análises significativas. Frequentemente, um erro pode fazer com que o tradutor altere drasticamente sua visão do programa-fonte, transformando o código correto em aparentemente incorreto em sua representação interna. Por exemplo, a omissão de uma única chave pode fazer com que muitas declarações válidas sejam tratadas como parte de um procedimento.

Vários métodos de análise sintática, tais como os métodos LL e LR, detectam erros tão cedo quanto possível. Mais precisamente, possuem a propriedade de prefixo viável, o que significa que detectam um erro quando o fluxo de tokens de entrada, lido da esquerda para a direita, não for o prefixo de qualquer cadeia da linguagem.

Exemplo 4.1. A fim de ser ter uma apreciação dos tipos de erros que ocorrem na prática, vamos examinar os erros que Ripley e Druseikis [1978] encontraram em uma amostra de programas Pascal de estudantes.

Ripley e Druseikis descobriram que os erros não ocorrem com muita frequência; 60% dos programas compilados estavam semanticamente e sintaticamente corretos. Mesmo quando os erros ocorreram, de fato, em média apenas 2,9 erros por programa incorreto, 80% dos encontrados consistiam em erros possíveis de serem reparados localmente. Em média, a maioria dos erros foram erros de pontuação, 60% envolveram erros em uma única linha.

Muitos desses erros poderiam ser classificados simplesmente como erros de pontuação. 29% de operadores e operandos, 15% de palavras-chave e mais 5% restantes de outros tipos. A grande maioria dos erros de pontuação ocorreu em torno do nível sintático de ponto e vírgula.

Para alguns erros comuns, considere o seguinte programa Pascal

```
(1)   ...
(2)   ...
(3)   ...
(4)   ...
(5)   ...
(6)   ...
(7)   ...
(8)   ...
(9)   ...

(10)  ...
(11)  ...
(12)  ...
(13)  ...
```

Um erro comum de pontuação é ter um ponto e vírgula em lugar de ponto e vírgula na lista de argumentos de uma declaração de função (por exemplo, usar ponto e vírgula em lugar do primeiro ponto e vírgula na linha 4) ou um ponto e vírgula em lugar de um ponto e vírgula no final de uma linha (por exemplo, ponto e vírgula em lugar do primeiro ponto e vírgula na linha 7). Um outro erro comum é ter um ponto e vírgula extra antes de uma palavra-chave else, por exemplo, ter um ponto e vírgula antes da linha 11.

Talvez uma razão pela qual esses erros de ponto e vírgula sejam tão comuns é que sua ocorrência varia grandemente de uma linguagem para outra. Em Pascal, um ponto e vírgula é um separador de comandos, e em PL/I é um terminador. Alguns usuários têm sugerido que alguma utilização de menos pesquisa seria benéfica [Conway e Wilcox 1973].

Um exemplo típico de um erro de operador é o de omitir todos os dois pontos da palavra-chave then. Não usualmente raros, mas outros erros de [illegible] serão um exemplo representativo.

Muitos compiladores Pascal têm dificuldades em tratar os erros comuns de inserção, omissão e transformação. De fato, vários compiladores Pascal não tratam corretamente o programa acima, provavelmente o erro mais comum de pontuação ou de operador, não emitir sequer uma mensagem de alerta apontando a construção ilegal.

Nos raros casos em que ocorre outro tipo de erro, pontuação, é difícil de ser reparado corretamente. É o caso de um [illegible] ausente (por exemplo, a omissão da linha 6). A maioria dos compiladores não faz nenhuma tentativa para esses tipos de erros. ▪

Como deve ser um tratador de erros ideal? No mínimo, deve-se esperar que a presença de um erro? No mínimo, deve-se informar o local no programa-fonte onde o

mento foi escolhido uma vez que existe uma boa chance de o erro efetivo ocorrer pouco antes do ponto onde o erro é detectado. Uma estratégia comum empregada por muitos compiladores é descartar a linha ilegal e imprimir uma apropriada para o programador, indicando a posição (linha e coluna) onde o erro foi detectado. Se existir um razoável prognóstico de que o erro realmente foi uma ocorrência de uma vírgula ausente nesta posição, uma mensagem de diagnóstico informativa também poderia ser fornecida, por exemplo, "ponto e vírgula ausente nesta posição?"

Uma vez que o erro tenha sido detectado, como deveria o analisador sintático se recuperar? Como veremos, existe um número de estratégias gerais, mas nenhuma delas claramente se impõe sobre as demais. Na maioria dos casos, não é adequado para o analisador sintático encerrar a execução depois de encontrar o primeiro erro, porque o processamento da entrada restante ainda pode revelar outros. Usualmente, envolve-se alguma forma de recuperação de erros na qual o analisador tenta restaurar a si mesmo para um estado onde o processamento da entrada possa continuar com uma razoável esperança de que o restante da entrada será analisado e tratado adequadamente pelo compilador.

Um trabalho inadequado de recuperação pode introduzir uma avalanche de erros "espúrios", que não foram cometidos pelo programador, mas introduzidos pelas modificações no estado do analisador sintático durante a recuperação de erros. Numa forma sutil, uma recuperação de erros sintáticos pode introduzir erros semânticos espúrios que serão detectados posteriormente pelas fases de análise semântica ou de geração de código. Por exemplo, ao se recuperar de um erro o analisador pode pular a declaração de algum identificador previamente declarado. Quando o identificador posteriormente for encontrado nas expressões, não haverá nada sintaticamente errado, mas como não há uma entrada na tabela de símbolos para ele, uma mensagem "variável não declarada" será gerada.

Uma estratégia excelente para o compilador é a de não dar mais mensagens de erro após detectar um erro, até que o analisador sintático tenha progredido uma certa distância à frente do ponto de detecção do último erro. Em alguns casos, pode haver erros demais para o compilador continuar um processamento sensível, e parece mesmo razoável que, depois de emitir um certo número de mensagens de diagnóstico (digamos, dez), o compilador desista de tentar encontrar mais erros. Isso evita que mensagens em cascata confundam o usuário. Em ambientes interativos de desenvolvimento de programas, onde o usuário quase sempre está sentado em frente a um terminal, pode ser útil dar ao usuário a opção de continuar a compilação após cada erro, ou após um certo número de erros, ou após um erro grave.

Como mencionamos, alguns compiladores tentam reparar os erros, num processo em que tentam adivinhar o que o programador queria escrever. O compilador PL/C (Conway e Wilcox [1973]) é um exemplo desse tipo. De fato, possivelmente uma audiência de pequenos programas escritos por estudantes principais a reparação extensiva de erros não é proposta a gerar o resultado. De fato, essa ênfase excessiva na complicação inteligente é uma das características de muitas técnicas para recuperação de erros que são mais propensas a atrapalhar do que a ajudar. Nossa ênfase aqui será em mecanismos simples de recuperação de erros.

Estratégias de Recuperação de Erros

Existem muitas estratégias gerais diferentes que um analisador sintático pode empregar para se recuperar de um erro sintático. Apesar de nenhuma delas ter provado ser universalmente aceitável, uns poucos métodos têm ampla aplicabilidade. Incluímos aqui as seguintes estratégias:

- modalidade do desespero
- nível de frase
- produções de erro
- correção global

Recuperação na modalidade do desespero. Este é o método mais simples de implementar e pode ser usado pela maioria dos métodos de análise sintática. Ao descobrir um erro, o analisador sintático descarta símbolos de entrada, um de cada vez, até que seja encontrado um *token* pertencente a um conjunto designado de tokens de *sincronização*. Os tokens de sincronização são usualmente delimitadores, tais como o ponto-e-vírgula ou o **end**, cujo papel no programa-fonte seja claro. Naturalmente, o projetista do compilador precisa selecionar os tokens de sincronização apropriados à linguagem-fonte. A vantagem da modalidade de desespero é que ela certamente pula uma parte considerável da entrada sem verificá-la, procurando por erros adicionais passar vantagem da simplicidade, e dificilmente cai em loops infinitos. Nas situações em que ocorrem múltiplos erros numa mesma declaração, esse método pode ser bastante adequado.

Recuperação em nível de frase. A recuperação em nível de frase pode realizar uma correção local na entrada restante. Isto pode substituir um prefixo da entrada remanescente por alguma cadeia que permita ao analisador sintático seguir em frente. Correções locais típicas podem substituir uma vírgula por um ponto e vírgula, remover um ponto-e-vírgula estranho ou inserir uma ausente. A escolha da correção local é deixada para o projetista do compilador. Naturalmente, devemos ser cuidadosos ao escolher substituições que não levem a laços infinitos, como seria o caso, por exemplo, se sempre inserisse-mos algo à frente do símbolo corrente.

Essa tipo de substituição pode ser usado em qualquer cadeia sem ter de lidar com situações complicadas de correção de erros. O método foi muito utilizado na análise sintática ascendente. Sua maior desvantagem está na dificuldade que tem ao lidar com situações nas quais o erro efetivo ocorreu antes do ponto de detecção.

Produções de erro. Se tivéssemos uma boa idéia dos erros comuns que poderiam ser encontrados, poderíamos aumentar a gramática para a linguagem em questão com as produções que gerassem construções ilegais. Usamos então a gramática aumentada com essas produções de erro para construir um analisador sintático. Se uma produção de erro for usada pelo analisador, poderemos gerar diagnósticos apropriados para rejeitar a construção ilegal que foi reconhecida na entrada.

Correção global. Idealmente, gostaríamos que um compilador fizesse tão poucas inferências quanto possível ao processar uma cadeia de entrada ilegal. Existem algoritmos para escolher uma sequência mínima de mudanças de forma a obter uma correção global de menor custo. Dadas uma cadeia de entrada incorreta x e uma gramática G, esses algoritmos irão encontrar uma árvore gramatical para uma cadeia relacionada y, de tal forma que as inserções, remoções e mudanças de tokens requeridas para transformar x em y sejam tão pequenas quanto possível. Infelizmente, esses métodos são em geral muito custosos de implementar, em termos de tempo e espaço, e então essas técnicas são atualmente apenas de interesse teórico.

Deve-se ressaltar que a programa correto mais próximo pode não ser aquele que o programador tinha em mente. Apesar de toda a noção de correção de custo mínimo fornecer um padrão de longo alcance para avaliar as técnicas de recuperação de erros, tem sido difícil para encontrar cadeias ótimas de substituição para a recuperação em nível de frase.

4.2 GRAMÁTICAS LIVRES DE CONTEXTO

Muitas construções de linguagens de programação possuem uma estrutura inerentemente recursiva que pode ser identificada por gramáticas livres de contexto. Por exemplo, poderíamos ter um comando condicional definido por uma regra tal como

> "se S_1 e S_2 são comandos então *if E* **then** S_1 **else** S_2 é um comando." (4.1)

Essa forma de comando condicional não pode ser especificada usando-se a notação para expressões regulares (no Capítulo 3), como vimos. As expressões regulares podem especificar a estrutura léxica dos tokens. Por outro lado, usando-se a variável sintática *cmd* para denotar a classe de comandos e *exp* para a classe de expressões, podemos prontamente expressar (4.1) usando a produção gramatical

$$cmd \rightarrow \textbf{if } exp \textbf{ then } cmd \textbf{ else } cmd \qquad (4.2)$$

Nesta seção, revisamos a definição de gramática livre de contexto e introduzimos a terminologia para falarmos a respeito da sua sintaxe. Na Seção 2.2, uma gramática livre de contexto era um

simplesmente consiste em terminais, não-terminais, um símbolo de partida e produções.

1. Os terminais são os símbolos básicos a partir dos quais as cadeias são formadas. A palavra "token" será um sinônimo de "terminal" no tocante a respeito de gramáticas para linguagens de programação. Em (4.2) cada uma das palavras-chave **if**, **then** e **else** é um terminal.
2. Os não-terminais são variáveis sintáticas que denotam conjuntos de cadeias. Em (4.2), **stmt** e **expr** são não-terminais. Os não-terminais definem conjuntos de cadeias que ajudam a definir a linguagem gerada pela gramática. Também impõem uma estrutura hierárquica na linguagem que é útil tanto para a análise sintática quanto para a tradução.
3. Numa gramática, um não-terminal é distinguido como o símbolo de partida, e o conjunto de cadeias que ele denota é a linguagem definida pela gramática.
4. As produções de uma gramática especificam a forma pela qual os terminais e não-terminais podem ser combinados a fim de formar cadeias. Cada produção consiste em um não-terminal, seguido por uma seta (às vezes o símbolo `::=` é usado no lugar da seta), seguido por uma cadeia de não-terminais e terminais.

Exemplo 4.2. A gramática com as seguintes produções define expressões aritméticas simples.

```
expr → expr op expr
expr → ( expr )
expr → - expr
expr → id
op → +
op → -
op → *
op → /
op → ↑
```

Nesta gramática, os símbolos terminais são

```
id + - * / ↑ ( )
```

Os símbolos não-terminais são `expr` e `op`, sendo `expr` o símbolo de partida.

Convenções Notacionais

A fim de evitar especificações cansativas com "terminais", "não-terminais" e assim por diante, iremos empregar as seguintes convenções notacionais relacionadas às gramáticas ao longo do resto deste livro.

1. Símbolos terminais:
   i) Letras minúsculas do início do alfabeto, tais como *a*, *b*, *c*.
   ii) Símbolos de operações, tais como +, *, etc.
   iii) Símbolos de pontuação, tais como parênteses, vírgula etc.
   iv) Os dígitos 0, 1, ..., 9.
   v) Cadeias em negrito como **id** ou **if**.
2. Símbolos não-terminais:
   i) Letras maiúsculas do início do alfabeto, tais como *A*, *B*, *C*.
   ii) A letra *S*, que, quando aparece, é usualmente o símbolo de partida.
   iii) Os nomes em italicos formados por letras maiúsculas, como *expr* ou *stmt*.
3. As letras maiúsculas do final do alfabeto, tais como *X*, *Y*, *Z*, representam *símbolos gramaticais*, isto é, terminais ou não-terminais.
4. Letras minúsculas, se for do interesse principalmente, tais como *t*, *u*, ..., *z*, representam cadeias de terminais.
5. Letras gregas minúsculas, α, β e γ, por exemplo, representam cadeias de símbolos gramaticais. Dessa forma, uma produção genérica poderia ser escrita como $A \to \alpha$, indicando que existe um único não-terminal A à esquerda da seta (o lado esquerdo da produção) e uma cadeia de símbolos gramaticais α à direita da seta (o lado direito da produção).
6. Se $A \to \alpha_1$, $A \to \alpha_2$, ..., $A \to \alpha_k$ são todas as produções com A à esquerda (chamamos de *produções-A*), podemos escrever $A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$. Chamamos $\alpha_1, \alpha_2, \ldots, \alpha_k$ de *alternativas* para A.
7. A menos que seja explicitamente estabelecido, o lado esquerdo da primeira produção é o símbolo de partida.

Exemplo 4.3. Usando as suas simplificações, podemos reescrever a gramática do Exemplo 4.2 convenientemente como

```
E → E A E | ( E ) | - E | id
A → + | - | * | / | ↑
```

Nessas convenções notacionais nos dizem que E e A são não-terminais, sendo E o símbolo de partida. Os símbolos restantes são terminais.

Derivações

Existem várias formas de se enxergar o processo pelo qual uma gramática define uma linguagem. Na Seção 2.2, examinamos este processo como sendo o de construir árvores gramaticais, mas existe também uma visão derivacional relacionada que é explicitamente orientada à tradução. De fato, a visão derivacional fornece uma precisa descrição da construção top-down da árvore gramatical. A idéia central aqui é que uma produção seja tratada como uma regra de reescrita, na qual o não-terminal à esquerda é substituído pela cadeia no lado direito da produção.

Por exemplo, considere a seguinte gramática para expressões aritméticas, com o não-terminal E representando uma expressão.

$$E \to E + E \mid E * E \mid (E) \mid - E \mid \mathbf{id} \tag{4.3}$$

A produção $E \to - E$ significa que uma expressão precedida por um sinal de menos também é uma expressão. Essa produção pode ser usada para gerar expressões mais complexas a partir de mais simples, permitindo-nos substituir qualquer instância de E por $- E$. No caso mais simples, podemos substituir uma única E por $- E$. Podemos descrever essa ação escrevendo

$$E \Rightarrow - E$$

que é lido "E deriva $- E$". A produção $E \to (E)$ nos diz que poderíamos também substituir uma instância de um E em qualquer cadeia de símbolos gramaticais por (E), por exemplo, $E + E \Rightarrow E + (E)$ ou $- E \Rightarrow - (E)$.

Podemos formalizar o conceito de derivação especificando as produções em qualquer ordem, a fim de obtermos uma sequência de substituições. Por exemplo,

$$E \Rightarrow - E \Rightarrow - (E) \Rightarrow - (E + E) \Rightarrow - (\mathbf{id} + E) \Rightarrow - (\mathbf{id} + \mathbf{id})$$

Chamamos uma tal sequência de substituições de uma *derivação* de $-(\mathbf{id} + \mathbf{id})$ a partir de E. Essa derivação provê a prova de que uma instância particular de uma expressão é a cadeia $-(\mathbf{id} + \mathbf{id})$.

No processamento mais abstrato, dizemos que se $\alpha A \beta \Rightarrow \alpha\gamma\beta$ e $A \to \gamma$ for uma produção e α e β forem cadeias arbitrárias de símbolos gramaticais. Se $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$, dizemos que α_1 *deriva* α_n. O símbolo $\Rightarrow$ significa "deriva em um passo". Frequentemente desejamos dizer "deriva em zero ou mais passos". Para esse propósito usamos o símbolo $\stackrel{*}{\Rightarrow}$. Dessa forma,

1. $\alpha \stackrel{*}{\Rightarrow} \alpha$ para qualquer cadeia α
2. Se $\alpha \stackrel{*}{\Rightarrow} \beta$ e $\beta \Rightarrow \gamma$, então $\alpha \stackrel{*}{\Rightarrow} \gamma$

Do mesmo modo, usamos $\stackrel{+}{\Rightarrow}$ para significar "deriva em um ou mais passos".

Dada uma gramática G, com símbolo de partida S, podemos usar a relação $\stackrel{*}{\Rightarrow}$ para definir $L(G)$, a *linguagem gerada por* G. As cadeias

...e em $L(G)$ se podem obter símbolos terminais de G. Dizemos que uma cadeia de terminais w está em $L(G)$ se e somente se $S \Rightarrow w$. A cadeia w é chamada de uma *sentença* de G. Uma linguagem que pode ser gerada por uma gramática é dita ser uma *linguagem livre de contexto*. Se duas gramáticas geram a mesma linguagem, as gramáticas são ditas *equivalentes*.

Se $S \Rightarrow \alpha$, onde α pode conter não terminais, dizemos, então, que α é uma *forma sentencial* de G. Uma sentença é uma forma sentencial despida de não terminais.

Exemplo 4.4. A cadeia $-(\mathbf{id}+\mathbf{id})$ é uma sentença da gramática 4.3 porque existe a derivação

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

As cadeias E, $-E$, $-(E)$, ..., $-(\mathbf{id}+\mathbf{id})$ ligando-se esta derivação, são todas formas sentenciais desta gramática. Escrevemos $E \Rightarrow -(\mathbf{id}+\mathbf{id})$ para indicar que $-(\mathbf{id}+\mathbf{id})$ pode ser derivada a partir de E.

Podemos mostrar por indução no comprimento de uma derivação que cada sentença desta linguagem é uma expressão aritmética envolvendo os operadores $+$, $-$, $*$, $/$, parênteses e o operando **id**. Similarmente, podemos mostrar por indução no comprimento de uma expressão aritmética que todas essas expressões podem ser geradas por essa gramática. Portanto, a gramática 4.3 gera precisamente o conjunto de todas as expressões aritméticas envolvendo os operadores binários $+$, $-$, $*$, $/$, unário $-$, parênteses e o operando **id**. ∎

A cada passo numa derivação, existem duas escolhas a serem feitas. Primeiro, precisamos escolher qual não terminal substituir a seguir, e, então, feita tal escolha, uma alternativa para a substituição daquele não terminal. Por exemplo, a derivação (4.4) do Exemplo 4.4 poderia ter começado a partir de $-(E+E)$, como se segue

$$-(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id}) \qquad (4.5)$$

Cada não-terminal em (4.5) é substituído pelo mesmo lado direito, como no Exemplo 4.4, mas a ordem das substituições é diferente.

Para compreender como certas análises sintáticas descendentes funcionam, precisamos considerar derivações nas quais somente o não-terminal mais à esquerda em qualquer forma sentencial seja substituído a cada passo. Tais derivações são ditas *derivações à esquerda*. Se $\alpha \Rightarrow \beta$ por um passo no qual o não-terminal mais à esquerda em α é substituído, escrevemos $\alpha \Rightarrow_{lm} \beta$. Como a derivação (4.4) é mais à esquerda, podemos reescrever a como

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$

Usando nossas convenções notacionais, cada passo numa derivação à esquerda pode ser escrito $wA\gamma \Rightarrow_{lm} w\delta\gamma$, onde w consiste somente de símbolos terminais, $A \to \delta$ é a produção aplicada, e γ é uma cadeia de símbolos gramaticais. Para enfatizar o fato de que α deriva β por uma derivação mais à esquerda, escrevemos $\alpha \Rightarrow_{lm} \beta$. Se $S \Rightarrow_{lm} \alpha$, dizemos, então, que α é uma *forma sentencial mais à esquerda* da gramática em questão.

Definições análogas valem para derivações mais à direita e não-terminais mais à direita em cada passo. Derivações mais à direita são, muitas vezes, chamadas de derivações canônicas.

Árvores Gramaticais e Derivações

Uma árvore gramatical pode ser vista como uma representação gráfica para uma derivação que difere somente na ordem de substituição. Relembrando, da seção 2.2, que cada nó interior de uma árvore gramatical é rotulado por algum não-terminal A e que os filhos desse nó são rotulados, da esquerda para a direita, pelos símbolos do lado direito da produção pela qual A foi substituído na derivação. As folhas da árvore gramatical são rotuladas por não-terminais ou terminais e, lidas da esquerda para a direita, constituem uma forma sentencial, chamada de produto ou fronteira da árvore. Por exemplo, a árvore gramatical para $-(\mathbf{id}+\mathbf{id})$ encontrada pela derivação (4.4) é mostrada na Fig. 4.2.

Fig. 4.2. Árvore gramatical para $-(\mathbf{id}+\mathbf{id})$.

Para se compreender a relação entre as derivações e as árvores gramaticais, consideremos uma derivação genérica $\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_{n-1} \Rightarrow \alpha_n \Rightarrow w$, onde w é uma cadeia terminal (lembre-se A). Para cada forma sentencial na derivação, construímos uma árvore gramatical com produto igual a α. O processo é uma indução em i. Como base da indução, a árvore para $\alpha_1 = A$ é um único nó rotulado A. Para realizar a indução, suponhamos já ter construído uma árvore gramatical cujo produto seja $\alpha_{i-1} = X_1 X_2 \cdots X_k$.

X_1 (Relembrando nossas convenções, cada X_i é um terminal ou não terminal.) Suponhamos que o próximo passo da derivação, $\alpha_{i-1} \Rightarrow \alpha_i$, seja obtido pela substituição de X_j, um não-terminal, por $\beta = Y_1 Y_2 \cdots Y_r$. Ou seja, o próximo passo da derivação, a produção $X_j \to \beta$ é aplicada a α_{i-1} a fim de derivar $\alpha_i = X_1 X_2 \cdots X_{j-1} \beta X_{j+1} \cdots X_k$.

Para modelar esse passo de derivação, encontramos a pesquisa folha a partir da esquerda na árvore gramatical construída. Esta folha é rotulada X_j. Damos a esta folha r filhos, rotulados $Y_1, Y_2, \ldots, Y_r$ a partir da esquerda. Como um caso especial, se $r = 0$, isto é, $\beta = \varepsilon$, então damos a pesquisa folha um filho rotulado ε.

Exemplo 4.5. Consideremos a derivação (4.4). A sequência de árvores gramaticais construída a partir desta derivação é mostrada na Fig. 4.3. No primeiro passo da derivação, $E \Rightarrow -E$. Para modelar este pas-

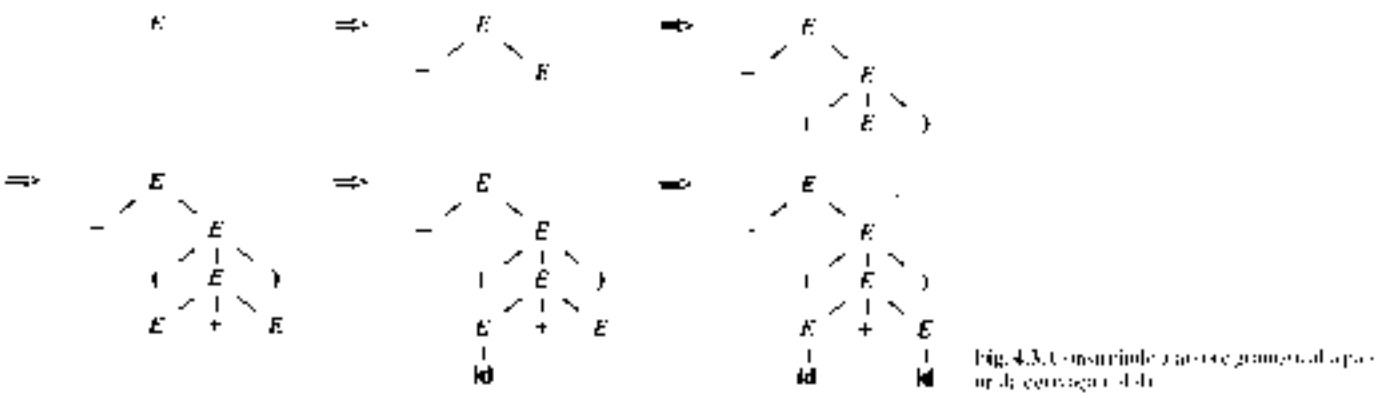


Fig. 4.3. Construindo a árvore gramatical para a derivação (4.4).

selecionamos dois filhos rotulados com E, à raiz E da árvore criada [Fig. 4.4(a)] e criamos a segunda árvore.

No segundo passo da derivação, $E \Rightarrow E * E$, novamente nós expandimos a folha mais à esquerda, criando três filhos rotulados com E, $*$ e E, à folha mais à direita de nossa árvore parcial [Fig. 4.4(b)]. Quando terminamos, produzimos a sentença de entrada id + id * id. Somente agora é que obtivemos uma árvore gramatical completa.

Como mencionamos, uma árvore gramatical ignora as variações na ordem pela qual os símbolos terminais são substituídos. Por exemplo, a derivação da Fig. 4.4 poderia também ser

$$E \Rightarrow E + E \Rightarrow E + E * E \Rightarrow \mathbf{id} + E * E \Rightarrow \mathbf{id} + \mathbf{id} * E \Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$$

Essas duas derivações, e muitas outras, produzem árvores gramaticais idênticas. Nosso algoritmo de análise sintática escolhe uma derivação mais à esquerda, mas as outras escolhas também podem ser implementadas. Em vez disso, é importante observar que cada árvore gramatical possui associada a si uma única derivação mais à esquerda e uma mais à direita. Na sequência, iremos frequentemente analisar sintaticamente através da reconstituição de uma derivação mais à esquerda ou mais à direita, sobretudo porque, como veremos, percorrer essas derivações é uma propriedade fundamental em muitos analisadores sintáticos. Note, entretanto, que nem toda sentença precisa necessariamente ser gerada por uma gramática ambígua. Uma gramática não ambígua também pode ter mais de uma derivação mais à esquerda ou mais à direita para uma dada sentença.

Exemplo 4.6. Vamos reescrever a gramática de expressões aritméticas do Ex. 4.1 de modo a produzir duas derivações distintas mais à esquerda para a sentença id + id * id. As duas derivações são:

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow \mathbf{id} + E \\ &\Rightarrow \mathbf{id} + E * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned} \qquad \begin{aligned} E &\Rightarrow E * E \\ &\Rightarrow E + E * E \\ &\Rightarrow \mathbf{id} + E * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * E \\ &\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned}$$

Uma das duas diversas gramáticas correspondentes encontra-se na Fig. 4.5.

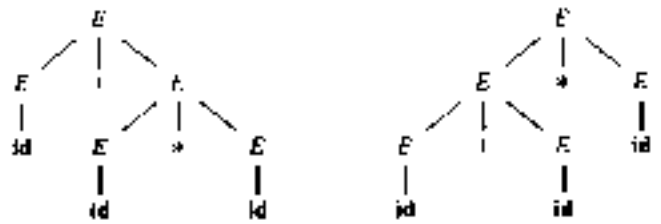


Fig. 4.4. Duas árvores gramaticais para id + id * id.

Note-se que a árvore gramatical da Fig. 4.4(a) reflete a precedência comumente assumida de $*$ sobre $+$, enquanto que a árvore da Fig. 4.4(b) não a reflete. Ou seja, estamos acostumados a pensar em $*$ como tendo uma precedência maior do que $+$, correspondendo ao fato de que normalmente avaliamos a expressão $a + b * c$ como $a + (b * c)$, e não como $(a + b) * c$.

Ambigüidade

Uma gramática que produz mais de uma árvore gramatical para alguma sentença é dita *ambígua*. Colocado de outra forma, uma gramática ambígua é aquela que produz mais de uma derivação mais à esquerda, ou mais à direita, para a mesma sentença. Para certos tipos de analisadores sintáticos, é desejável que a gramática seja inambígua, porque senão não poderemos selecionar, de forma única, uma árvore gramatical para uma dada sentença. Para algumas aplicações, também veremos métodos através dos quais podemos usar gramáticas ambíguas, porém com regras de *desambigüização* que "descartam" certas gramáticas, restando apenas uma única árvore para cada sentença.

4.3 ESCREVENDO UMA GRAMÁTICA

As gramáticas são capazes de descrever quase todas as construções de linguagens de programação. Uma parte inicial da análise sintática é realizada pelo analisador léxico, na medida em que produz uma sequência de tokens a partir dos caracteres de entrada. Certas restrições feitas a entidades como identificadores, que não podem ser descritas por uma gramática livre de contexto. Consequentemente, as seqüências de tokens aceitas por um analisador sintático formam um subconjunto de uma linguagem de programação, sendo subseqüentemente necessário analisar a sua saída a fim de assegurar a correção semântica das construções que não são facilmente analisadas sintaticamente (veja o Capítulo 6).

Começamos esta seção considerando a divisão de trabalho entre um analisador léxico e um analisador sintático. Depois, uma questão básica de análise sintática pode ser a de gramáticas que incluam uma certa transformação. A gramática inicial pode ter que ser reescrita a fim de torná-la analisável pelo método escolhido. As gramáticas adequadas às vezes são difíceis de serem construídas usando-se nossas técnicas de análise sintática preferidas. Entretanto, como veremos na Seção 2.4, nem toda gramática é adequada a uma análise sintática preditiva. Concluímos esta seção considerando algumas construções de linguagens de programação que não podem ser descritas por qualquer gramática.

Expressões Regulares vs. Gramáticas Livres de Contexto

Cada construção que possa ser descrita por uma expressão regular também pode ser descrita por uma gramática. Por exemplo, a expressão regular $(a|b)^*abb$ sobre o alfabeto $\{a, b\}$ é a linguagem reconhecida pelo AFN da Fig. 3.23. Uma gramática correspondente, construída a partir do AFN, é a seguinte. Para cada estado i do AFN, criamos um símbolo não-terminal A_i. Se o estado i possui uma transição para o estado j sob a entrada a, introduzimos a produção $A_i \rightarrow aA_j$. Se o estado i vai para o estado j sob a entrada ε, introduzimos a produção $A_i \rightarrow A_j$. Se i é um estado de aceitação, introduzimos $A_i \rightarrow \varepsilon$. Se i é o estado de partida, fazer de A_i o símbolo de partida da gramática.

Uma vez que cada expressão regular é uma linguagem livre de contexto, podemos razoavelmente indagar "por que usar expressões regulares para definir certa classe de linguagem?" Existem várias razões.

1. As regras léxicas de uma linguagem são frequentemente muito simples, e para descrever os símbolos não necessitamos de uma notação tão poderosa como as gramáticas.
2. As expressões regulares geralmente provêem uma notação mais concisa e facilmente compreendida para os tokens do que as gramáticas.
3. A partir de expressões regulares podem ser construídos automaticamente analisadores léxicos mais eficientes do que os que podem ser feitos a partir de gramáticas arbitrárias.
4. A separação da estrutura sintática de uma linguagem em partes léxicas e não-léxicas provê uma maneira conveniente de modularizar a vantagem de um compilador em componentes administrativamente gerenciáveis.

Não existem diretrizes formalmente estabelecidas sobre o que colocar nas regras léxicas, ao contrário das regras sintáticas. As expres-

são regulares são muito úteis para descrever a estrutura de construções léxicas tais como identificadores, constantes, palavras-chave e assim por diante. Por outro lado, as gramáticas são muito úteis na descrição de estruturas aninhadas tais como parênteses balanceados, begin-end correspondentes, if-then-else correspondentes e assim por diante. Contudo, uma vez estabelecidas tais estruturas aninhadas nos podem ser descritas por expressões regulares.

Verificando a Linguagem Gerada por uma Gramática

Apesar de os projetistas de compiladores raramente efetuarem provas formais de uma linguagem de programação completa é importante estarmos capacitados a raciocinar quanto a um dado conjunto de produções gera uma linguagem particular. Às vezes construções problemáticas podem ser descobertas ao se tentar escrever uma gramática sucinta e concisa e em seguida analisando-se a linguagem que ela mesma gera. Adiante, vemos exemplos de uma gramática para expressões envolvendo operadores.

Uma prova de que uma gramática G gera uma linguagem L possui duas partes: precisamos mostrar que cada cadeia gerada por G está em L, e, reciprocamente, que toda cadeia em L pode ser gerada por G.

Exemplo 4.7. Considere a gramática

$$S \rightarrow (S)S \mid \varepsilon \tag{4.6}$$

Pode não ser evidente à primeira vista que esta gramática simples gera todas as cadeias de parênteses balanceados e somente tais cadeias. Para verificarmos isso, mostraremos primeiro que toda sentença derivável de S é balanceada, e em seguida que cada cadeia balanceada é derivável a partir de S. Para mostrar que cada sentença derivável de S é balanceada usamos uma prova indutiva sobre o número de passos numa derivação. Como base da indução, nota-se que a única cadeia de terminais derivável a partir de S em um único passo é a cadeia vazia, a qual certamente é balanceada.

Vamos assumir agora que todas as derivações com menos de n passos produzam sentenças balanceadas e consideremos uma derivação com n passos. Tal derivação deve ser da forma

$$S \Rightarrow (S)S \Rightarrow \cdots \Rightarrow (x)y$$

As derivações de x e y a partir de S têm menos que n passos e, então, pela hipótese indutiva, x e y são balanceadas. Dessa forma, a cadeia $(x)y$ tem que ser balanceada.

Mostramos, então, que qualquer cadeia derivável a partir de S é balanceada. Precisamos mostrar em seguida que qualquer cadeia balanceada é derivável a partir de S. Para realizar isso, usamos a indução no comprimento de uma cadeia. Como base da indução, a cadeia vazia é derivável a partir de S.

Vamos assumir agora que toda cadeia balanceada de comprimento menor que $2n$, onde $2n$ seja derivável a partir de S, e consideremos uma cadeia balanceada de comprimento $2n$, $n \geq 1$. Certamente a cadeia começa por um parêntese à esquerda. Seja (x) o menor prefixo de w tendo um número igual de parênteses à esquerda e à direita. Então, w pode ser escrita como $(x)y$, onde ambas x e y são balanceadas. Como x e y são de comprimento menor que $2n$, são deriváveis a partir de S pela hipótese indutiva. Dessa forma, podemos encontrar uma derivação da forma

$$S \Rightarrow (S)S \Rightarrow \cdots \Rightarrow (x)y$$

provando que $w = (x)y$ também é derivável a partir de S. □

Eliminando a Ambigüidade

Algumas vezes uma gramática ambígua pode ser reescrita de forma a eliminar a ambigüidade. Como um exemplo, vamos eliminar a ambigüidade da seguinte gramática do "then-else":

$$\begin{aligned} cmd &\rightarrow \textbf{if } expr \textbf{ then } cmd \\ &\mid \textbf{if } expr \textbf{ then } cmd \textbf{ else } cmd \\ &\mid outro \end{aligned} \tag{4.7}$$

Aqui "outro" significa qualquer outro comando. De acordo com esta gramática o comando condicional aninhado

$$\textbf{if } E_1 \textbf{ then } S_1 \textbf{ else if } E_2 \textbf{ then } S_2 \textbf{ else } S_3$$

possui a árvore gramatical mostrada na Fig. 4.5. A gramática (4.7) é ambígua porque não fica claro se a cadeia

$$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2 \tag{4.8}$$

possui as duas árvores mostradas na Fig. 4.6.

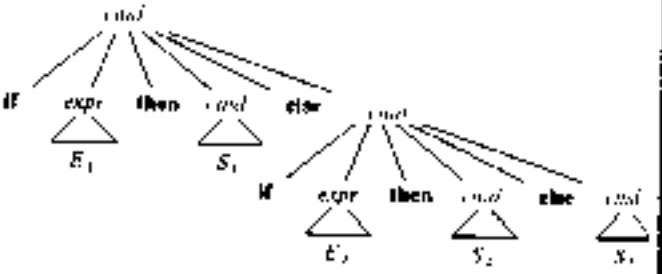


Fig. 4.5 Árvore gramatical para o comando condicional.

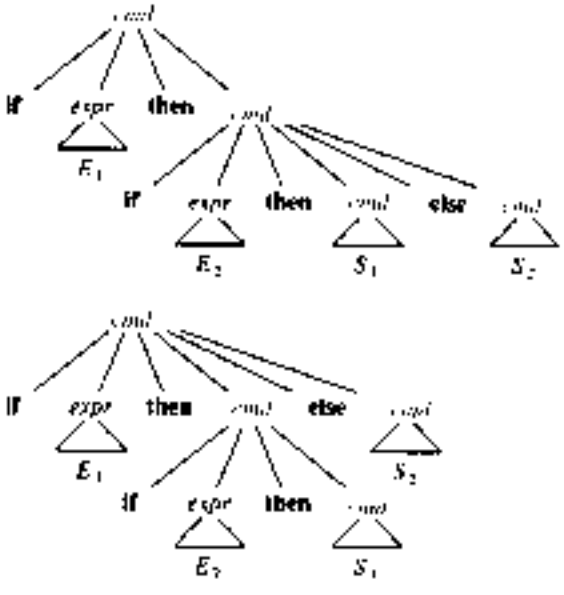


Fig. 4.6 Duas árvores gramaticais para uma sentença ambígua.

Em todas as linguagens de programação com comandos condicionais desta forma, a primeira árvore gramatical é preferida. A regra geral é "associe cada else ao then anterior mais próximo ainda não associado". Essa regra de não ambigüidade pode ser incorporada diretamente à gramática. Por exemplo, podemos reescrever a gramática (4.7) sob a forma não ambígua abaixo. A idéia é que um comando fechado é aquele que contém um **then** e um **else** pareados (isto é, não pode terminar com um **then** aberto). Um comando fechado também pode ser qualquer comando não condicional. Um comando aberto é aquele que tem um **then** não associado. Um comando associado é um comando **if-then-else** contendo somente comandos associados ou é qualquer outro tipo de comando incondicional. Dessa forma podemos usar a gramática

```
cmd → call id ( lista_expr )
    | id [ lista_expr ] = expr
    | if expr then cmd
    | if expr then cmd else cmd
    | while expr do cmd
    | begin lista_cmd end
lista_expr → lista_expr , expr | expr
lista_cmd → lista_cmd ; cmd | cmd
```

Esta gramática gera o mesmo conjunto de cadeias que a Fig. 4.7, mas permite uma estruturação gramatical para cada especificação, especificamente aquela que associa cada **else** com o **then** anterior mais próximo não associado.

Eliminação da Recursão à Esquerda

Uma gramática é recursiva à esquerda se possui um não-terminal A tal que exista uma derivação $A \Rightarrow A\alpha$ para alguma cadeia α. Os métodos de análise sintática descendente não podem processar gramáticas recursivas à esquerda e, consequentemente, uma transformação que elimine a recursão à esquerda é necessária. Na Seção 2.4, discutimos a recursão simples à esquerda, onde há uma produção da forma $A \rightarrow A\alpha | \beta$. Aqui, estudaremos algoritmos mais gerais. Na Seção 2.4, mostramos como o par de produções recursivas à esquerda $A \rightarrow A\alpha | \beta$ poderia ser substituído pelas produções não-recursivas

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \varepsilon \end{aligned}$$

sem mudar o conjunto de cadeias de caracteres derivadas a partir de A. Esta regra por si mesma é suficiente para muitas gramáticas.

Exemplo 4.8. Considere a seguinte gramática para expressões aritméticas:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned} \tag{4.09}$$

Uma tradução recursiva descendente para (4.09) poderia ter procedimentos para as três não-terminais E, T e F. Como $E \Rightarrow E + T$ e $T \Rightarrow T * F$, o procedimento para E e T pode chamar a si mesmo sem que um símbolo de entrada seja consumido. Uma gramática como (4.09) é dita ser recursiva à esquerda porque possui um não-terminal A tal que $A \Rightarrow A\alpha$ para alguma cadeia α. Um método de análise descendente, incluindo o preditivo, pode entrar em um laço infinito para uma gramática recursiva à esquerda a menos que seja tomado cuidado especial durante o projeto do analisador sintático.

Nossa primeira tarefa é mostrar como eliminar a recursão à esquerda de uma gramática. Começaremos com o caso em que não existem produções da forma $A \rightarrow A\alpha$, isto é, nenhuma produção cujo lado direito comece com o lado esquerdo. Em seguida, mostraremos como isso pode ser feito em geral, sem a preocupação com produções da forma $A \rightarrow A\alpha$. O algoritmo 4.1 elimina sistematicamente a recursão à esquerda de uma gramática. É garantido funcionar se a gramática não possui ciclos (derivações da forma $A \Rightarrow A$) ou produções-ε (produções da forma $A \rightarrow \varepsilon$). Ciclos podem ser eliminados de uma gramática da mesma forma que as produções-ε (ver Exercícios 4.20 e 4.22).

Algoritmo 4.1. Eliminação da recursão à esquerda.

Entrada. Uma gramática G sem ciclos ou produções-ε.

Saída. Uma gramática equivalente sem recursão à esquerda.

Método. Aplique o algoritmo na Fig. 4.9 a G. Note que a gramática resultante não-recursiva à esquerda pode ter produções-ε. □

1. Coloque os não-terminais em alguma ordem $A_1, A_2, \ldots, A_n$.
2. **para** $i := 1$ **até** n **faça início**
   para $j := 1$ **até** $i - 1$ **faça início**
      substituir cada produção da forma $A_i \rightarrow A_j\gamma$
      pelas produções $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$
      onde $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ são todas as gramáticas-A_j correntes
   fim
   eliminar as recursões imediatas à esquerda entre as produções-A_i
fim

Fig. 4.9. Algoritmo para eliminar a recursão à esquerda de uma gramática.

A razão pela qual o procedimento da Fig. 4.9 funciona é que após a i-ésima iteração do laço **para** mais externo no passo 2, qualquer produção da forma $A_k \rightarrow A_l\alpha$, onde $k < i$, terá necessariamente $l > k$. Consequentemente, a iteração seguinte do laço externo, no máximo, progressivamente aumenta o limite inferior em p para qualquer produção da forma $A_i \rightarrow A_j\alpha$. Assim que tivermos $m = i - 1$, então a eliminação da recursão imediata à esquerda para as produções-A_i obrigará p a ser i ou mais, toda vez que i.

Exemplo 4.9. Vamos aplicar este procedimento à gramática (4.11). Tecnicamente, o algoritmo 4.1 não é garantido funcionar porque há produções-ε, mas, neste caso particular, a produção $T \rightarrow \varepsilon$ é inofensiva.

Ordenamos os não-terminais A, B, C. Não existe recursão imediata à esquerda entre as produções-A, consequentemente, nada acontece durante o passo 2 para o caso $i = 1$. Para $i = 2$, substituímos as produções-A em $B \rightarrow BA \mid b$ a fim de obter as seguintes produções-B:

$$B \rightarrow Bda \mid Bbc \mid ba \mid bc$$

Eliminando a recursão imediata à esquerda entre as produções-B, para a seguinte gramática:

$$\begin{aligned} B &\rightarrow baB' \mid bcB' \\ B' &\rightarrow daB' \mid bcB' \mid \varepsilon \end{aligned}$$

Fatoração à Esquerda

A fatoração à esquerda é uma transformação gramatical útil para a criação de uma gramática adequada à análise sintática preditiva. A idéia básica está em quando não estiver claro qual das duas produções alternativas usar para expandir um não-terminal A, estaremos capacitados a reescrever as produções-A de postergar a decisão até que tenhamos visto o suficiente da entrada para fazer uma escolha correta.

Por exemplo, se tivermos as duas produções

```
cmd → if expr then cmd else cmd
    | if expr then cmd
```

...expressão condicional e ainda if-then-else, podemos incluir também a versão mais geral possível, escolhendo a forma de expandir A em Lugar-se $A \to \alpha\beta$, são duas produções. Uma vez que não sabemos se devemos expandir A em α ou β, fazemos as duas coisas. Isto é, criamos uma nova variável A' e reescrevemos as produções A como $A \to \alpha A' \mid \beta A'$. Em seguida, podemos expandir A' em α ou β, se for o caso. Isto é, as produções originais, fatoradas à esquerda, se tornam

$$A \to \alpha A'$$
$$A' \to \beta \mid \gamma$$

Algoritmo 4.2. Eliminação à esquerda de uma gramática.

Entrada: Gramática G.

Saída: Uma gramática equivalente fatorada à esquerda.

Método: Para cada não-terminal A, encontre o maior prefixo α comum a duas ou mais de suas alternativas. Se $\alpha \neq \varepsilon$, existe uma produção não-recursiva $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, onde γ representa todas as alternativas que não começam com α. Substitua por

$$A \to \alpha A' \mid \gamma$$
$$A' \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

Aqui A' é uma nova variável. Aplique repetidamente esta transformação até que não haja duas alternativas com um prefixo comum.

Exemplo 4.10. A seguinte gramática abstrata modela uma construção *if-then-else*:

$$S \to \mathbf{if}\ E\ \mathbf{then}\ S \mid \mathbf{if}\ E\ \mathbf{then}\ S\ \mathbf{else}\ S \mid a \qquad (4.13)$$
$$E \to b$$

Aqui E é a expressão condicional, *if* e *then* são terminais, S é a instrução "expressão" e a representa "outras instruções". Esta forma abstrata é útil para evitar a ambiguidade (4.13). Ela também pode ser fatorada à esquerda:

$$S \to \mathbf{if}\ E\ \mathbf{then}\ S\ S' \mid a \qquad (4.14)$$
$$S' \to \mathbf{else}\ S \mid \varepsilon$$
$$E \to b$$

Nessa forma, podemos expandir S' para "else S" ou para ε, o que for apropriado para a entrada. Por exemplo, para expandir S' para "else S", escolhemos $S' \to \mathbf{else}\ S$. Naturalmente, as gramáticas (4.13) e (4.14) são ambíguas e a ação apropriada para uma cadeia de entrada "else" é ainda mais obscura. Na Seção 4.3 veremos como lidar com ambiguidades. Por exemplo, a Regra 1 do Exemplo 4.19 diz que uma forma de se usar essa definição é sempre associar "else" com o "then" anterior mais próximo. ■

Construções de Linguagens Não Livres de Contexto

Não devemos nos surpreender com o fato de algumas linguagens não possuírem gramáticas livres de contexto. De fato, algumas poucas construções simples encontradas em muitas linguagens de programação não podem ser especificadas de maneira alguma usando-se gramáticas. Nesta seção, apresentamos várias dessas construções, usando linguagens abstratas simples para ilustrar as dificuldades.

Exemplo 4.11. Considere a linguagem abstrata $L_1 = \{wcw \mid w \text{ é uma cadeia de } a\text{'s e } b\text{'s}\}$. L_1 consiste de todas as palavras compostas de uma cadeia de a's e b's repetidas, separadas por um c (as mesmas cadeias à esquerda e à direita). Pode ser provado que essa linguagem não é livre de contexto. Esta linguagem abstrata modela o problema de verificar se identificadores são declarados antes de serem usados. Uma cadeia w antes do c representa a declaração de um identificador, e uma w depois do c representa o uso de um identificador. A linguagem L_1 em si não representa exatamente o problema, porque não exige que os identificadores sejam declarados antes de serem usados. Também, ela não descreve a limitação de comprimento de identificadores. O fato de que L_1 é uma abstração de problemas de linguagens de programação reais justifica um estudo de sua não-liberdade de contexto. Exemplos de linguagens como L_1 que descrevem declarações de identificadores antes de seus usos e que permitem identificadores de comprimento arbitrário...

Devemos ressaltar que gramáticas para a estrutura de blocos aninhados, como as de Pascal, são possíveis, mesmo quando os nomes dos identificadores não são limitados. As construções de uma linguagem de programação em que um nome deve ser declarado antes de ser usado são mais difíceis de serem detectadas. Muitos compiladores usam uma tabela de símbolos para guardar informações sobre identificadores. Os nomes são inseridos na tabela quando são declarados. Quando um nome é encontrado em uma instrução executável, verifica-se se ele está na tabela de símbolos. Se não estiver, é gerada uma mensagem de erro. ■

Exemplo 4.12. A linguagem $L_2 = \{a^n b^m c^n d^m \mid n \geq 1 \text{ e } m \geq 1\}$ é livre de contexto. Um exemplo de problema que é modelado de maneira semelhante à de L_2 é o de chamadas de procedimentos, em que o número de parâmetros reais deve ser igual ao número de parâmetros formais. Suponha que as cadeias a^n e b^m representem as listas de parâmetros formais em duas declarações de procedimentos, e que c^n e d^m representem as listas de parâmetros reais em duas chamadas desses procedimentos. Isto é, estamos interessados em saber se $n = m$, onde n é o número de parâmetros formais e m é o número de parâmetros reais em uma chamada. Novamente, não estamos interessados em saber se os parâmetros reais são do tipo correto. Especificações mais realistas também são modeladas de maneira semelhante.

Devemos mencionar que a sintaxe típica das declarações e usos de procedimentos não se preocupa em verificar o número de parâmetros. Por exemplo, o código de uma linguagem como C consiste de uma sequência de definições e comandos de procedimentos, com chamadas de procedimentos da forma

```
id ( expr , expr , ... , expr )
```

para produções adequadas para *expr*. A verificação de que o número de parâmetros reais está correto em uma chamada é tratada durante a fase de análise semântica. ■

Exemplo 4.13. A linguagem $L_3 = \{ww \mid w \text{ é uma cadeia de } a\text{'s e } b\text{'s}\}$ não é livre de contexto. Um exemplo de problema que é modelado de maneira semelhante à de L_3 é o de chamadas de procedimentos em que não limitamos o número de parâmetros. Lembre-se de que a expressão regular $(a \mid b)^*a(a \mid b)^*a(a \mid b)^*$ descreve todas as cadeias de a's e b's que contêm pelo menos dois a's, mas não especifica a posição dos a's. Um problema semelhante ocorre quando queremos saber se uma cadeia de entrada representando uma lista de parâmetros formais é permitida para uma chamada de procedimento, com uma cadeia representando a lista de parâmetros reais. Não podemos escrever uma gramática livre de contexto que faça essa verificação, a menos que limitemos o tamanho das cadeias que podem ser nomes de parâmetros. Em vez disso, a linguagem L_3 representa as palavras simplificadas. A cadeia $a^n b^m a^n b^m$ representa uma chamada de procedimento com n parâmetros reais em ambas as chamadas e m parâmetros reais em cada uma. Novamente, não estamos interessados em saber se os parâmetros reais são do tipo correto. Uma gramática para L_3 é dada no Exemplo 4.17. ■

É interessante notar que linguagens muito semelhantes a L_1, L_2 e L_3 são livres de contexto. Por exemplo, $L'_1 = \{wcw^R \mid w \text{ é uma cadeia de } a\text{'s e } b\text{'s}\}$, onde w^R é w escrita ao contrário, é gerada pela gramática

$$S \to aSa \mid bSb \mid c$$

A linguagem $L'_2 = \{a^n b^m c^m d^n \mid n \geq 1 \text{ e } m \geq 1\}$ é livre de contexto, gerada por gramática

$$S \to aSd \mid aAd$$
$$A \to bAc \mid bc$$

Igualmente, $L'_3 = \{ww^R \mid w \text{ é uma cadeia de } a\text{'s e } b\text{'s}\}$ é livre de contexto, gerada por gramática

$$S \to aSa \mid bSb \mid M$$
$$M \to aMa \mid bMb \mid \varepsilon$$

Finalmente, $L'_4 = \{a^n b^m \mid n \geq m \geq 1\}$ é livre de contexto, gerada por gramática

$$S \to aSb \mid ab$$

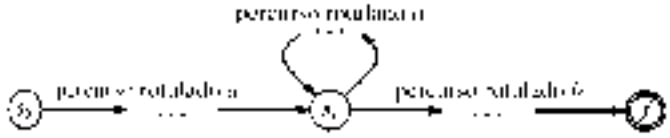


Fig. 4.8. AFD D tal que $L(D) = a\alpha^*b$.

É importante notar que L_{ab} é o primeiro de vários exemplos de uma linguagem infinita definida por qualquer expressão regular. Para estudar mais esses resultados, vamos supor uma linguagem definida por algum autômato finito determinístico D qualquer. Vamos supor também que D possua n estados e que a entrada x tenha n ou mais símbolos. Considere a sequência de estados $s_0, s_1, \ldots, s_n$ pelos quais D passa ao ler os n primeiros símbolos de x. Como D possui somente n estados diferentes, pelo menos dois estados na sequência $s_0, s_1, \ldots, s_n$ precisam ser o mesmo estado. Digamos que $s_j = s_k$ e que $j < k$. Ou seja, o estado s_j é o primeiro estado que D entra mais de uma vez ao ler os símbolos de x.

Agora, k deve ser pelo menos 1, pois não há dois estados iniciais. Também k deve ser no máximo n, pois D entra em apenas n estados após ler os n primeiros símbolos de x. Portanto, s_j é um estado que D entra pela primeira vez após ler o j-ésimo símbolo de x e também após ler o k-ésimo símbolo de x. Isso significa que D também aceita xy, onde y consiste em $k - j$ cópias consecutivas dos símbolos de x entre os símbolos $j + 1$ e k. Em outras palavras, D também aceita xy^2, xy^3 e, de fato, xy^i para todo $i \geq 0$.

Como D possui somente n estados diferentes, pelo menos dois destes estados na sequência $s_0, s_1, \ldots, s_n$ precisam ser os mesmos, digamos s_j e s_k. A partir do estado s_j, uma sequência de $k - j$ leva D a um estado de aceitação, uma vez que o estado s_k é também s_j. Mas também existe uma sequência de entrada mais curta que leva D do estado s_j a um estado de aceitação, a saber, a sequência obtida ao se remover a parte do estado s_j até s_k (ou seja, a sequência $x_{k+1} x_{k+2} \ldots x_n$). Dessa forma, D também aceita a cadeia xz que possui $j + (n - k)$ símbolos, contradizendo a suposição de que D é a linguagem aceita por D.

Coloquialmente, dizemos que "um autômato finito não pode realizar contagens", significando que não pode aceitar uma linguagem como L_{ab} que exigiria que o autômato mantivesse uma contagem do número de a's desde que encontrou o a inicial. Similarmente, dizemos que uma expressão regular não pode manter uma contagem de dois ou mais pois de três, mas vemos que com a gramática podemos definir L_{ab}, por exemplo.

4.4 ANÁLISE SINTÁTICA *TOP-DOWN*

Nesta seção, introduzimos as duas técnicas para análise sintática que consideramos e mostramos como construir uma forma eficiente de analisador sintático que deve ser corretamente chamado de analisador sintático preditivo. Determinamos a classe de gramáticas LL(1) a partir da qual os analisadores preditivos podem ser construídos automaticamente. Além de formalizar a discussão dos analisadores preditivos da Seção 2.4, consideramos aqui os analisadores sintáticos preditivos não recursivos. Esta seção conclui com uma discussão da recuperação de erros. Os analisadores sintáticos *bottom-up* são discutidos nas Seções 4.5 a 4.7.

Análise Sintática de Descendência Recursiva*

A análise sintática top-down pode ser vista como uma tentativa de se encontrar uma derivação mais à esquerda para uma cadeia de entrada. Equivalentemente, pode ser vista como uma tentativa de se construir uma árvore gramatical para a cadeia de entrada a partir de nós e arcos mais do topo (a árvore gramatical é explicada na Seção 2.2). Na Seção 2.4, discutimos um caso especial da análise sintática de descendência recursiva onde não era necessário retrocesso. Descrevemos agora uma forma geral de análise sintática top-down, chamada de descendência recursiva, que pode envolver retrocesso, ou seja, a realização de repetidos deslocamentos da entrada. Por outro lado, esses analisadores sintáticos com retrocesso raramente são usados muito frequentemente. Uma razão é que o retrocesso é raramente necessário para analisar construções de linguagens de programação, e as situações nas quais a análise sintática de linguagens naturais requer retrocesso pode frequentemente ser evitada (ahoCorasick, 1975). Para compiladores de programação de finalidade geral, o método de Earley [1970] são preferidos. Ver Aho e Ullman [1972b] para uma descrição desses métodos gerais de análise sintática.

O retrocesso é exigido no próximo exemplo, e iremos sugerir uma forma de controlar a entrada quando o retrocesso ocorrer.

---

*Esta seção pode ser omitida em uma primeira leitura.

Exemplo 4.14. Considere a gramática

$$S \rightarrow cAd \quad (4.13)$$
$$A \rightarrow ab \mid a$$

e a cadeia de entrada $w = cad$. Para construir uma árvore gramatical para esta cadeia de cima para baixo, começamos especificando uma árvore consistindo de um único nó rotulado S, a especificação inicial da gramática. Com a primeira entrada c sendo lida, a única produção para S a fim de expandir a árvore é a (4.13), e obtemos a Fig. 4.9(a).

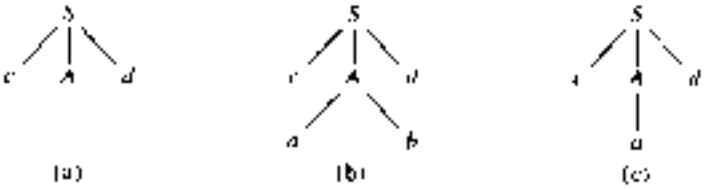


Fig. 4.9. Etapas na construção de uma árvore gramatical.

A folha mais à esquerda, rotulada c, corresponde ao primeiro símbolo de w, de modo que podemos avançar o apontador da entrada para a segunda posição de w, a, e considerar o nó rotulado A em seguida. Expandindo A usando a sua primeira alternativa, obtemos a árvore da Fig. 4.9(b). Temos agora um nó rotulado a para o segundo símbolo da entrada e, consequentemente, avançamos o apontador da entrada para d, o terceiro símbolo da entrada, e comparamos d com a próxima folha rotulada b. Como b não casa com d, retrocedemos pela folha e retornamos A a fim de verificar se existe uma outra alternativa que não tentamos ainda, mas que poderia possibilitar um resultado melhor.

Ao retornarmos de volta para A, precisamos redefinir o apontador da entrada para a posição 2, aquela que o mesmo depois que passamos pela primeira vez por A, o que significa que o procedimento para A é análogo ao procedimento para não-terminais na Fig. 2.17(c) precisa armazenar o apontador da entrada numa variável local. Tentamos agora a segunda alternativa de A, a fim de obter a árvore na Fig. 4.9(c). A folha a reconhece o segundo símbolo de w, e a folha d reconhece o último símbolo. Uma vez que produzimos uma árvore gramatical para w, sabemos que a análise sintática termina com sucesso.

Uma gramática recursiva à esquerda pode levar um analisador sintático de descendência recursiva, mesmo com retrocesso, a um laço infinito. Isto é, quando tentamos expandir A, podemos eventualmente nos encontrar de novo tentando expandir A sem ter consumido nenhum símbolo da entrada.

Analisadores Sintáticos Preditivos

Em muitos casos, ao projetar um analisador sintático de descendência recursiva, podemos evitar o retrocesso escolhendo a alternativa correta para cada A que deve ser expandida. Se formos bem-sucedidos em projetar uma gramática com essa propriedade, então podemos construir um analisador sintático preditivo recursivo, como discutido na Seção 2.4. Para construir um analisador sintático preditivo, precisamos conhecer, dado o símbolo corrente de entrada a e o não-terminal A a ser expandido, qual das alternativas da produção $A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$ é a única que deriva uma cadeia começando por a. Ou seja, a alternativa adequada precisa ser detectada examinando-se apenas o primeiro símbolo da cadeia que a mesma deriva. As construções de controle de fluxo em muitas linguagens de programação, com suas palavras chaves distintivas, são normalmente detectáveis desta forma. Por exemplo, se tivermos as produções

$$\textit{cmd} \rightarrow \textbf{if}\ \textit{expr}\ \textbf{then}\ \textit{cmd}\ \textbf{else}\ \textit{cmd}$$
$$\mid\ \textbf{while}\ \textit{expr}\ \textbf{do}\ \textit{cmd}$$
$$\mid\ \textbf{begin}\ \textit{lista\_de\_comandos}\ \textbf{end}$$

e marcas polares usadas em *while* e *begin*-*end* nos informam qual alternativa deve ser aplicada, caso possível tendo em vista o próximo símbolo de entrada.

Diagramas de Transições para Analisadores Sintáticos Preditivos

Na Seção 2.4.4, discutimos a implementação de analisadores sintáticos preditivos através de procedimentos recursivos, como, por exemplo, aqueles da Fig. 2.17. Lá afirmamos que o diagrama de transição seguinte foi considerado uma planta ou fluxograma útil para um analisador léxico. Podemos criar um diagrama de transições como um plano ou fluxograma útil para um analisador sintático preditivo.

As várias diferenças entre os diagramas de transições para um analisador léxico e um analisador sintático preditivo são imediatamente aparentes. Num analisador sintático, existe um diagrama para cada não-terminal. Os símbolos dos rótulos são tokens e não-terminais. Uma transição em um token terminal significa que devemos avançar a entrada se aquele token for o próximo símbolo da entrada. Uma transição em um não-terminal é uma chamada do procedimento para ele.

Para construir um diagrama de transições de um analisador sintático preditivo a partir de uma gramática, eliminamos primeiro da gramática a recursividade à esquerda, e em seguida fatoramos à esquerda. Para cada não-terminal A, executamos os seguintes passos:

1. Criamos um estado inicial e um final (de aceitação).
2. Para cada produção $A \rightarrow X_1 X_2 \cdots X_n$, criamos uma percurso a partir do estado inicial até o estado final, com os rótulos rotulados $X_1, X_2, \ldots, X_n$.

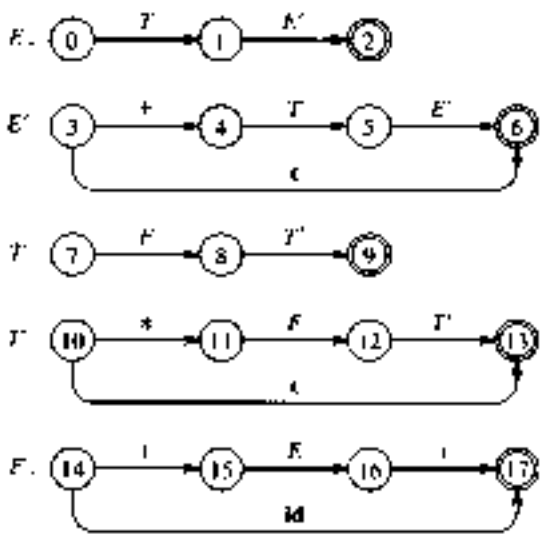


Fig. 4.16. Diagramas de transições para expressões.

O analisador preditivo irá trabalhar sobre os diagramas de transições se comporta como segue. Começa no estado inicial para o símbolo não-terminal de partida. Se após algumas ações estiver no estado s, e o próximo símbolo de entrada for a, e existir uma transição do estado s para o estado t rotulada a, então o analisador avança o cursor da entrada e vai para o estado t. Se em vez disso a transição rotulada a não existir, o analisador simplesmente executa como se o estado s fosse um estado de aceitação. Em cada estado, o próximo passo é determinado pelos rótulos das transições que saem daquele estado e pelo próximo símbolo de entrada. Uma transição em um símbolo terminal a é usada se a for o próximo símbolo da entrada. Uma transição em um não-terminal é uma chamada do procedimento para A.

Para construir um diagrama de transições de um analisador sintático preditivo a partir de uma gramática, eliminamos primeiro da gramática a recursividade à esquerda, e em seguida fatoramos à esquerda. Para cada não-terminal A, executamos os seguintes passos:

1. Criamos um estado inicial e um final (de aceitação).
2. Para cada produção $A \rightarrow X_1 X_2 \cdots X_n$, criamos uma percurso a partir do estado inicial até o estado final, com os rótulos rotulados $X_1, X_2, \ldots, X_n$.

O analisador preditivo irá trabalhar sobre os diagramas de transições se comporta como segue. Começa no estado inicial para o símbolo não-terminal de partida. Se após algumas ações estiver no estado s, e o próximo símbolo de entrada for a, e existir uma transição do estado s para o estado t rotulada a, então o analisador avança o cursor da entrada e vai para o estado t. Se em vez disso a transição rotulada a não existir, o analisador simplesmente executa como se o estado s fosse um estado de aceitação. Em cada estado, o próximo passo é determinado pelos rótulos das transições que saem daquele estado e pelo próximo símbolo de entrada. Uma transição em um símbolo terminal a é usada se a for o próximo símbolo da entrada. Uma transição em um não-terminal é uma chamada do procedimento para ele. Quando o estado de aceitação é encontrado, o procedimento retorna.

Opcionalmente, podemos fundir estados de diagramas. Devemos ter cuidado para não criar um diagrama de transições que leve a um analisador sintático preditivo ambíguo. Em geral, não existem regras de uma vez para determinar se fundir estados irá criar um diagrama ambíguo, mas, conceitualmente, podemos fundir estados sempre que houver arestas rotuladas com o mesmo símbolo saindo de estados distintos, mas tais estados devem ser evitados sempre que houver uma escolha a ser feita entre duas arestas com o mesmo rótulo. Veremos um exemplo em breve.

Discutiremos a implementação dos diagramas de transições em mais detalhes logo em seguida.

A substituição em diagramas de transições de um não-terminal por seu diagrama pode produzir diagramas não determinísticos, isto é, pode existir mais de uma transição de um estado para outros ao ler uma entrada. Se a ambiguidade resultante levar a erros, estaremos diante de uma gramática não LL(1). Em geral, não existe uma regra simples para eliminar a não determinação. Se o diagrama determinístico não puder ser eliminado, não poderemos construir um analisador sintático preditivo recursivo para a gramática. Se não puder ser construído um analisador de descendente recursivo não-backtracking, isto é, um que possa prever sistematicamente qual produção aplicar, mesmo com a ajuda de um número arbitrário de símbolos de entrada à frente, não existe nenhuma melhor estratégia de análise que possamos empregar.

Exemplo 4.15. A Fig. 4.16 contém uma coleção de diagramas de transições para a gramática (4.1)–(4.4). As setas e rótulos correspondem exatamente às regras da gramática. Se tivéssemos substituído os rótulos E' e T' pelos diagramas para E' e T', teríamos um diagrama para E com muitas setas saindo do estado inicial. Também teríamos duas setas saindo do estado inicial para T, uma rotulada com F e outra com $($. Se, além disso, soubermos que o próximo símbolo de entrada deve ser $(id$ ou $($, não existe nenhuma escolha a ser feita entre essas duas setas, pois uma leva a um estado de onde id deve ser o próximo símbolo de entrada, enquanto a outra leva a um estado de onde $($ deve ser o próximo símbolo de entrada. Assim, podemos construir um analisador preditivo recursivo para a gramática original, embora não para a gramática (4.1)–(4.4). ☐

Os diagramas de transições podem ser simplificados pela substituição dos diagramas uns pelos outros. Essas substituições são semelhantes às transformações feitas nas gramáticas estudadas na Seção 2.5. Por exemplo, na Fig. 4.16(c), a chamada de E' foi substituída pelo corpo do diagrama para E'.

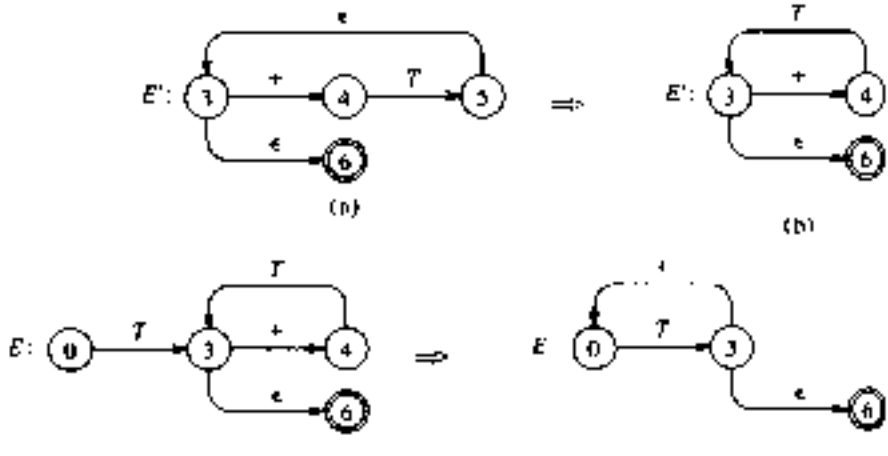


Fig. 4.17. Diagramas de transições simplificados.

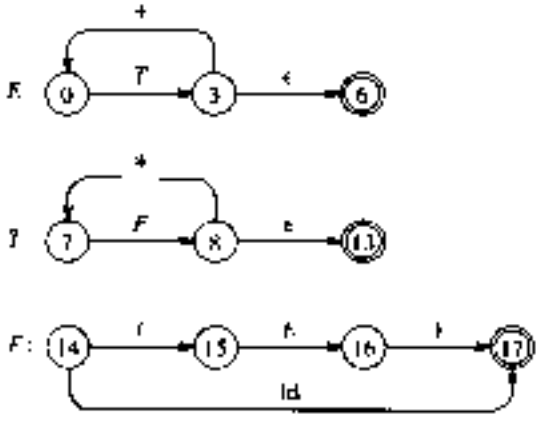


Fig. 4.12. Diagrama de transições simplificado para expressões aritméticas.

A Fig. 4.11 (b) mostra um diagrama de transições equivalente para E'. Podemos substituir o diagrama da Fig. 4.11 (b) pela transição em E' no diagrama para E na Fig. 4.10, produzindo o diagrama da Fig. 4.11 (c). Por fim, observamos que os primeiro e terceiro nós na Fig. 4.11 (c) são equivalentes e os combinamos. O resultado é a Fig. 4.11 (d), repetido, sem os primeiro diagrama da Fig. 4.12. As outras duas etapas são análogas para T e T'. O conjunto completo de diagramas resultantes é mostrado na Fig. 4.12. Uma implementação C para esse analisador preditivo roda de 2/3 a 2/5 mais rapidamente do que uma implementação C para o da Fig. 4.10.

Análise Sintática Preditiva Não-Recursiva

É possível construir um analisador preditivo descendente não-recursivo mantendo explicitamente uma pilha ao invés de implicitamente através de chamadas recursivas. O problema chave durante a análise preditiva é determinar que produção deve ser aplicada a um dado não-terminal. O analisador não-recursivo da Fig. 4.14 prevenu pela produção a ser aplicada consultando a tabela explícita. Veremos como a tabela pode ser construída diretamente a partir de certas gramáticas.

Um analisador sintático preditivo dirigido por uma tabela mostra um *buffer* de entrada, uma pilha, uma tabela sintática e um fluxo de saída. O *buffer* de entrada contém a cadeia a ser analisada, seguida por um símbolo de fim-de-cadeia usado para indicar o fim da cadeia de entrada. A pilha contém uma sequência de símbolos gramaticais, com $ usado como fundo da pilha. Inicialmente, a pilha contém o símbolo de partida da gramática acima de $. Uma tabela sintática é um array bidimensional $M[A, a]$, onde A é um não-terminal e a é um terminal ou o símbolo especial $.

O analisador sintático é controlado por um programa que se comporta como segue. O programa considera X o símbolo do topo da pilha e a o símbolo corrente de entrada. Esses dois símbolos determinam a ação do analisador. Existem três possibilidades:

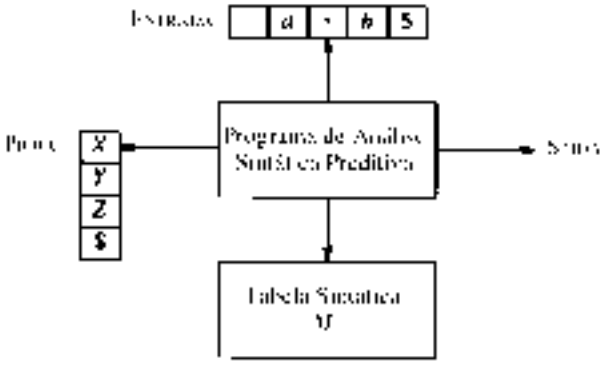


Fig. 4.13. Modelo de um analisador sintático preditivo não-recursivo.

1. Se $X = a = \$$, o analisador pára e anuncia término com sucesso da análise sintática.
2. Se $X = a \neq \$$, o analisador sintático remove X da pilha e avança o apontador da entrada para o próximo símbolo.
3. Se X é um não-terminal, o programa consulta a entrada $M[X, a]$ da tabela sintática M. Essa entrada será uma produção de X da gramática ou uma entrada de erro. Se, por exemplo, $M[X, a] = \{X \to UVW\}$, o analisador substitui X no topo da pilha por WVU (com U no topo). Como saída, iremos assumir que o analisador imprime a produção usada; de fato, qualquer outro código poderia ser executado aqui. Se $M[X, a] =$ erro, o analisador chama uma rotina de recuperação de erros.

O comportamento do analisador sintático pode ser descrito em termos de suas configurações, que dão o conteúdo da pilha e o entrada restante.

Algoritmo 4.3. Análise sintática preditiva não-recursiva.

Entrada. Uma cadeia w e uma tabela sintática M para a gramática G.

Saída. Se w está em $L(G)$, uma derivação mais à esquerda de w; caso contrário, uma indicação de erro.

Método. Inicialmente, o analisador sintático está em uma configuração na qual possui S na pilha com S o símbolo de partida de G no topo, e $w\$$ no *buffer* de entrada. O programa que utiliza a tabela sintática preditiva M para realizar uma análise sintática descendente é mostrado na Fig. 4.14.

```
faça ip apontar para o primeiro símbolo de w$;
repetir
        seja X o símbolo-gramática do topo da pilha e a o símbolo apontado por ip;
        se X for um terminal ou $ então
                se X = a então
                        remover X do topo da pilha e avançar ip
                senão erro()
        senão /* X é um não-terminal */
                se M[X, a] = X → Y1 Y2 ... Yk então início
                        remover X do topo da pilha;
                        empilhar Yk, Yk-1, ..., Y1 sobre o topo da pilha;
                        escrever a produção X → Y1 Y2 ... Yk
                fim
                senão erro()
até que X = $      /* a pilha está vazia */
```

Fig. 4.14. Programa de análise sintática preditiva.

Exemplo 4.16. Considere a gramática (4.1)–(4.3) do Exemplo 4.8. Uma tabela sintática preditiva para essa gramática é mostrada na Fig. 4.15. Os brancos são entradas de erro; não-brancos indicam uma produção com a qual se deve expandir o símbolo não-terminal no topo da pilha. Note que também discutiremos como essas entradas podem ser selecionadas, porém por ora vejamos o funcionamento do

Com a entrada **id + id * id**, o analisador preditivo realiza a sequência de movimentos na Fig. 4.16. O apontador da entrada aponta para o símbolo mais à esquerda da cadeia na coluna ENTRADA. Se observarmos cuidadosamente as ações deste analisador, veremos que ele mantém (a menos de símbolos no topo da pilha) a forma da esquerda da entrada. As produções escolhidas são aquelas de uma derivação mais à esquerda. Os símbolos de entrada que já foram explicitamente seguidos pelos símbolos gramaticais na pilha desaparecem para deixar a forma sentencial mais à esquerda na derivação.

PRIMEIRO e SEGUINTE

A construção de um analisador sintático preditivo é auxiliada por duas funções associadas à gramática G. Essas funções, PRIMEIRO e SE-

NÃO-TERMINAL	SÍMBOLO DE ENTRADA					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

Fig. 4.18. Tabela sintática M para a gramática (4.1).

PILHA	ENTRADA	SAÍDA
$E$$	**id** + **id** * **id**$	
$E'T$$	**id** + **id** * **id**$	$E \to TE'$
$E'T'F$$	**id** + **id** * **id**$	$T \to FT'$
$E'T'$**id**	**id** + **id** * **id**$	$F \to$ **id**
$E'T'$	+ **id** * **id**$	
E'	+ **id** * **id**$	$T' \to \varepsilon$
$E'T+$	+ **id** * **id**$	$E' \to +TE'$
$E'T$	**id** * **id**$	
$E'T'F$	**id** * **id**$	$T \to FT'$
$E'T'$**id**	**id** * **id**$	$F \to$ **id**
$E'T'$	* **id**$	
$E'T'F*$	* **id**$	$T' \to *FT'$
$E'T'F$	**id**$	
$E'T'$**id**	**id**$	$F \to$ **id**
$E'T'$	$	
E'	$	$T' \to \varepsilon$
$	$	$E' \to \varepsilon$

Fig. 4.16. Movimentos feitos pelo analisador sintático preditivo para a cadeia **id** + **id** * **id**.

SEGUINTE nos permitem preencher as entradas de uma tabela sintática preditiva para a gramática, sempre que possível. Os comandos de *ação de erro* podem ser preenchidos pela função SEGUINTE, podendo também ser usados como auxílio de sincronização durante a recuperação de erros na modalidade do modo de pânico.

Se α for qualquer cadeia de símbolos gramaticais, seja PRIMEIRO(α) o conjunto de terminais que começam as cadeias derivadas a partir de α. Se $\alpha \Rightarrow^* \varepsilon$, então ε também está em PRIMEIRO(α).

Definimos SEGUINTE(A), para o não-terminal A, como sendo o conjunto de terminais a que podem ligar imediatamente à direita de A em uma forma sentencial, isto é, o conjunto de terminais a tais que existam uma derivação de forma $S \Rightarrow^* \alpha A a \beta$, para alguns α e β. Note que podem ter existido, em algum tempo durante a derivação, símbolos entre A e a, mas se isso ocorreu, os mesmos derivaram ε e desapareceram. Se A puder ser o símbolo mais à direita em alguma forma sentencial, então $ está em SEGUINTE(A).

Para computar PRIMEIRO(X) para todos os símbolos gramaticais X, aplique as seguintes regras até que nenhum terminal ou ε possa ser adicionado a qualquer conjunto PRIMEIRO.

1. Se X for um terminal, então PRIMEIRO(X) = $\{X\}$.

2. Se $X \to \varepsilon$ for uma produção, adicione ε a PRIMEIRO(X).

3. Se X for um não-terminal e $X \to Y_1 Y_2 \ldots Y_k$ uma produção, coloque a em PRIMEIRO(X) se, para algum i, a estiver em PRIMEIRO(Y_i) e ε estiver em todos PRIMEIRO(Y_1), ..., PRIMEIRO(Y_{i-1}); isto é, se $Y_1 \ldots Y_{i-1} \Rightarrow^* \varepsilon$. Se ε estiver em PRIMEIRO(Y_j) para todo $j = 1, 2, \ldots, k$, adicione ε a PRIMEIRO(X). Por exemplo, tudo o que estiver em PRIMEIRO(Y_1) estará certamente em PRIMEIRO(X). Se Y_1 não derivar ε, então não adicionamos mais nada a PRIMEIRO(X), mas se $Y_1 \Rightarrow^* \varepsilon$, então precisamos adicionar PRIMEIRO(Y_2), e assim por diante.

Agora podemos computar PRIMEIRO para qualquer cadeia $X_1 X_2 \ldots X_n$ como segue. Adicione a PRIMEIRO($X_1 X_2 \ldots X_n$) todos os símbolos não-ε de PRIMEIRO(X_1). Adicione, também, todos os símbolos não-ε de PRIMEIRO(X_2), se ε estiver em PRIMEIRO(X_1); todos os símbolos não-ε de PRIMEIRO(X_3), se ε estiver em PRIMEIRO(X_1) e PRIMEIRO(X_2); e assim por diante. Finalmente, adicione ε a PRIMEIRO($X_1 X_2 \ldots X_n$) se, para todo i, PRIMEIRO(X_i) contiver ε.

Para computar SEGUINTE(A) para todos os não-terminais A, aplique as seguintes regras até que nada mais possa ser adicionado a qualquer conjunto SEGUINTE.

1. Coloque $ em SEGUINTE(S), onde S é o símbolo de partida e $ o marcador de fim de entrada à direita.

2. Se existir uma produção $A \to \alpha B \beta$, então tudo em PRIMEIRO(β), exceto ε, é colocado em SEGUINTE(B).

3. Se existir uma produção $A \to \alpha B$ ou uma produção $A \to \alpha B \beta$ onde PRIMEIRO(β) contém ε (isto é, $\beta \Rightarrow^* \varepsilon$), então, tudo em SEGUINTE(A) está em SEGUINTE(B).

Exemplo 4.17. Consideremos a gramática (4.1), repetida abaixo.

$$
\begin{aligned}
E &\to TE' \\
E' &\to +TE' \mid \varepsilon \\
T &\to FT' \\
T' &\to *FT' \mid \varepsilon \\
F &\to (E) \mid \mathbf{id}
\end{aligned}
$$

Então

PRIMEIRO(E) = PRIMEIRO(T) = PRIMEIRO(F) = {(, **id**}.

PRIMEIRO(E') = {+, ε}

PRIMEIRO(T') = {*, ε}

SEGUINTE(E) = SEGUINTE(E') = {), $}

SEGUINTE(T) = SEGUINTE(T') = {+,), $}

SEGUINTE(F) = {+, *,), $}

Por exemplo, **id** e o parêntese à esquerda são adicionados a PRIMEIRO(T) pela regra 3 na definição de PRIMEIRO, com $i = 1$ em cada caso, uma vez que PRIMEIRO(**id**) = {**id**} e PRIMEIRO(F) = {(, **id**} pela regra 1. Pela regra 3), com $i = 1$, a produção $E \to TE'$ implica que **id** e o parêntese à esquerda estão em PRIMEIRO(E) igualmente. Como um exemplo, $ está em PRIMEIRO(E') pela regra 2).

Para computar os conjuntos SEGUINTE, colocamos $ em SEGUINTE(E) pela regra 1) para SEGUINTE. Pela regra 2), aplicada à produção $F \to (E)$, o parêntese direito está em SEGUINTE(E). Pela regra 1), aplicada à produção $E \to TE'$, $ é colocado em SEGUINTE(E'). Como $E' \to \varepsilon$ também está em SEGUINTE(E'). Como um último exemplo de como as regras para SEGUINTE podem ser aplicadas, a produção $E \to TE'$ implica, pela regra 2), que tudo que não seja ε em PRIMEIRO(E') precisa ser colocado em SEGUINTE(T). Já vimos que + está em SEGUINTE(T). □

Construção de Tabelas Sintáticas Preditivas

O próximo algoritmo pode ser usado para construir uma tabela sintática preditiva para uma gramática G. A idéia por trás do algoritmo é a seguinte: suponhamos que $A \to \alpha$ seja uma produção com a em PRIMEIRO(α). Então, eventualmente, o analisador sintático irá expandir A através de α quando o símbolo de entrada corrente for a. A única complicação ocorre quando $\alpha = \varepsilon$ ou $\alpha \Rightarrow^* \varepsilon$. Nesse caso, deveríamos expandir A de novo através de α se o símbolo corrente de entrada estiver em SEGUINTE(A) ou se o fim da entrada tiver sido atingido e SEGUINTE(A) contiver $\$$.

Algoritmo 4.4. Construção de uma tabela sintática preditiva.

Entrada: Uma gramática G.

Saída: Tabela sintática M.

Método:

1. Para cada produção $A \to \alpha$ da gramática, execute os passos 2 e 3.
2. Para cada terminal a em PRIMEIRO(α), adicione $A \to \alpha$ a $M[A, a]$.
3. Se ε estiver em PRIMEIRO(α), adicione $A \to \alpha$ a $M[A, b]$, para cada terminal b em SEGUINTE(A). Se ε estiver em PRIMEIRO(α) e $\$$ em SEGUINTE(A), adicione $A \to \alpha$ a $M[A, \$]$.
4. Faça cada entrada indefinida de M ser **erro**.

Exemplo 4.18. Vamos aplicar o algoritmo 4.4 à gramática (4.1). Como PRIMEIRO(E') = PRIMEIRO(T) = {(, id}, a produção $E \to TE'$ faz com que $M[E, (] = E \to TE'$ e $M[E, \text{id}] = E \to TE'$ adquiram a entrada $E \to TE'$.

A produção $E' \to +TE'$ causa $M[E', +] = E' \to +TE'$. A produção $E' \to \varepsilon$ causa $M[E',)] = E' \to \varepsilon$ e $M[E', \$] = E' \to \varepsilon$, uma vez que SEGUINTE($E'$) = {), $}.

A tabela sintática preditiva dada pelo algoritmo 4.4 para a gramática (4.1) foi mostrada na Fig. 4.15.

Gramáticas LL(1)

O algoritmo 4.4 pode ser aplicado a qualquer gramática G para produzir uma tabela sintática M. Para muitas gramáticas, entretanto, algumas entradas serão multiplamente definidas. Por exemplo, se G for recursiva à esquerda ou ambígua, M terá pelo menos uma entrada multiplamente definida.

Exemplo 4.19. Vamos considerar a gramática (4.13) do Exemplo 4.10 de novo a mesma é repetida aqui por uma questão de conveniência.

$$S \to iEtS \mid iEtSeS \mid a$$
$$E \to b$$

A tabela sintática para esta gramática é mostrada na Fig. 4.17.

A entrada para $M[S', e]$ contém $S' \to eS$ e $S' \to \varepsilon$ uma vez que SEGUINTE(S') = {e, $}. A gramática é ambígua e a ambigüidade é manifestado pela escolha sobre qual produção usar quando um **else** for encontrado. Podemos resolver a ambigüidade se escolhermos $S' \to eS$. Esta escolha corresponde a associar os **else**'s aos **then**'s anteriores mais próximos. Note-se que a escolha $S' \to \varepsilon$ impedirá para sempre que **e** viesse a ser colocado na pilha ou removido da entrada e, por conseguinte, está seguramente errada.

Uma gramática cuja tabela sintática não possui entradas multiplamente definidas é dita LL(1). O primeiro "L" em LL(1) significa a varredura da entrada da esquerda para a direita (do inglês *left to right scan*), o segundo "L" significa a produção de uma derivação mais à esquerda (do inglês *leftmost derivation*) e o "1" o uso de um único símbolo de entrada como *lookahead* a cada passo para tomar as decisões sintáticas. Pode ser mostrado que o algoritmo 4.4 produz para cada gramática LL(1) G, uma tabela sintática que descreve todas e somente as sentenças de G.

As gramáticas LL(1) possuem várias propriedades distintas. Nenhuma gramática ambígua ou recursiva à esquerda pode ser LL(1). Pode também ser mostrado que uma gramática G é LL(1) se e somente se, sempre que $A \to \alpha \mid \beta$ forem duas produções distintas de G, as condições seguintes estiverem.

1. Se α e β não derivarem, ao mesmo tempo, cadeias começando pelo mesmo terminal a, qualquer que seja a.
2. No máximo um dos dois, α ou β, derivam a cadeia vazia.
3. Se $\beta \Rightarrow^* \varepsilon$, então α não deriva a qualquer cadeia começando por um terminal em SEGUINTE(A).

Claramente a gramática (4.13) para expressões aritméticas não é LL(1). A gramática (4.15), modelando construções *if-then-else*, também não é.

Resta a questão sobre o que deveria ser feito quando a tabela sintática possui entradas multiplamente definidas. Um recurso é reescrever a gramática através da eliminação de toda a recursividade à esquerda e à fatoração à esquerda sempre que possível na esperança de produzir uma gramática para a qual a tabela sintática não possua entradas multiplamente definidas. Infelizmente, existem algumas gramáticas para as quais nenhuma alteração irá produzir uma gramática LL(1). A gramática (4.13) e muitas outras linguagens não possuem gramática LL(1). Como vimos, podemos ainda decompor (4.13) através de um analisador sintático preditivo fazendo alterações em $M[S', e]$. Em geral, não existem regras conhecidas pelas quais as entradas multiplamente definidas possam ser transformadas univocamente definidas sem alterar a linguagem reconhecida pelo analisador sintático.

A dificuldade principal em se usar a análise preditiva está na escrita de uma gramática para a linguagem fonte tal que um analisador sintático preditivo possa ser construído a partir da mesma. Apesar da eliminação da recursividade à esquerda e da fatoração à esquerda serem fáceis de aplicar, ambas tornam a gramática resultante difícil de ser usada para os fins da tradução. Para analisar algumas dessas dificuldades, uma forma de organização comum para um front-end sintático de um compilador está em se usar um analisador preditivo para construções de controle e usar a precedência de operadores, discutida na Seção 4.6, para as expressões. Ferramentas como um gerador de analisadores sintáticos LR estão disponíveis, como discutido na Seção 4.9, podendo-se obter mais particularmente os benefícios da análise preditiva e de precedência de operadores.

Recuperação de Erros na Análise Preditiva

A pilha de um analisador preditivo não recursivo torna explícitos os terminais e não-terminais que o mesmo espera reconhecer como construção da entrada. Iremos conseqüentemente nos referir aos símbolos na pilha do analisador na discussão que se segue. Um erro é detectado durante a análise preditiva quando o terminal no topo da pilha não corresponde ao próximo símbolo de entrada ou quando o não-terminal A está no topo da pilha, a é o próximo símbolo de entrada e a entrada da tabela sintática $M[A, a]$ está vazia.

Não-terminal	Símbolo de entrada					
	a	b	e	i	t	$\$$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to eS$ $S' \to \varepsilon$			$S' \to \varepsilon$
E		$E \to b$				

Fig. 4.17. Tabela sintática M para a gramática (4.13).

A recuperação de erros na modalidade de desempenho está baseada na ideia de se ignorar entradas, a partir daquela que surgiu primeiramente estando o conjunto SEGUINTE do símbolo não terminal sendo desempilhado. Suponha que o analisador sintático encontre uma entrada que não pode ser emparelhada com o topo da pilha. Uma possibilidade é desempilhar o topo da pilha e prosseguir a análise. Isso pode ser razoável se parecer provável que a entrada não vista foi gerada pelo símbolo não terminal que acabou de ser desempilhado. Se o topo da pilha for um símbolo terminal, uma possibilidade é emitir uma mensagem de erro dizendo que o símbolo terminal foi inserido e prosseguir a análise sintática. Na prática, essa técnica pode causar muitos problemas, a menos que a maioria dos erros possíveis tenha sido considerada cuidadosamente ao se projetar o analisador sintático.

Uma técnica mais sofisticada para a recuperação de erros em um analisador sintático preditivo é baseada em se inserir ou modificar símbolos na entrada, de modo a corrigir erros comuns. Uma possibilidade é examinar a entrada com base em pares de símbolos. Dado o símbolo atual na entrada e o símbolo no topo da pilha, o analisador sintático pode modificar a entrada ou desempilhar o topo da pilha. Uma questão importante é se podemos construir uma recuperação de erros razoável baseada apenas em pares de símbolos. Outra questão é se os erros reais serão tratados de forma razoável. Uma terceira questão é se essa estratégia é fácil de implementar.

Uma abordagem para a recuperação de erros em um analisador sintático preditivo é a de se atribuir cadeias de caracteres de erro a cada entrada da tabela sintática preditiva. Essas cadeias especificam a mensagem de erro a ser gerada, ou referenciam uma rotina a ser chamada quando um erro for detectado durante a consulta à tabela. Também podemos modificar a tabela sintática preditiva para incluir ponteiros para rotinas de recuperação de erros, que podem inserir ou modificar a entrada, ou desempilhar a pilha.

Recuperação em nível de frase. A recuperação em nível de frase é implementada preenchendo-se as entradas em branco da tabela sintática preditiva com ponteiros para rotinas de erro. Essas rotinas podem modificar, inserir ou apagar símbolos da entrada, e emitir as mensagens de erro apropriadas. Podem também remover símbolos da pilha. Uma questão importante é se devemos permitir a alteração dos símbolos da pilha, ou empilhar novos símbolos, uma vez que os passos dados pelo analisador podem não corresponder à derivação de palavra alguma da gramática. Em qualquer caso, devemos estar certos de que haja a possibilidade de um progresso. A verificação de que uma ação de recuperação resulte em um símbolo de entrada, sendo consumido, ou a pilha sendo encurtada se a entrada estiver vazia, é um princípio básico para impedir que um laço infinito ocorra.

4.5 ANÁLISE SINTÁTICA *BOTTOM-UP*

Nesta seção, introduzimos o estilo geral de análise sintática *bottom-up*, conhecida como análise de empilhar e reduzir. Uma forma fácil de implementá-la, chamada de análise sintática de precedência de operadores, é apresentada na Seção 4.6. Um método muito mais geral de análise de empilhar e reduzir, chamado de análise LR, é discutido na Seção 4.7. A ferramenta Yacc discutida na Seção 4.9 usa esses métodos de análise LR.

A análise gramatical de empilhar e reduzir tenta construir uma árvore gramatical para uma cadeia de entrada começando pelas folhas (o fundo) e trabalhando a árvore acima em direção à raiz (o topo). Podemos pensar neste processo como o de "reduzir" uma cadeia *w* ao símbolo de partida de uma gramática. A cada passo de redução, uma subcadeia particular que reconhecemos ser lado direito de uma produção é substituída pelo símbolo à esquerda daquela produção, e se a subcadeia for escolhida corretamente a cada passo, uma derivação mais à direita reversa será traçada em ordem inversa.

Exemplo 4.21. Considere a gramática

$$S \rightarrow aABe$$
$$A \rightarrow Abc \mid b$$
$$B \rightarrow d$$

Passo	Entrada	Comentário
(1)	id + id * id$	erro, pular +
(2)	id + id * id$	id está em PRIMEIRO(E)
(3)	id + id * id$	
(4)	id + id * id$	
(5)	id + id * id$	
(6)	id + id * id$	
(7)	+ id * id$	
(8)	+ id * id$	erro, M[E, *] = sinc
(9)	+ id * id$	E' foi removido do topo da pilha
(10)	+ id * id$	
(11)	+ id * id$	
(12)	+ id * id$	
(13)	id * id$	
(14)	id * id$	
(15)	id * id$	
(16)	id * id$	
(17)	* id$	
(18)	* id$	
(19)	id$	
(20)	id$	
(21)	$	
(22)	$	
(23)	$	

Fig. 4.19. Movimentos da análise sintática e recuperação de erros feitos sobre a entrada errada id + id * id.

A cadeia de entrada id + id * id é analisada sintaticamente com a ação de recuperação de erros da Fig. 4.19 sendo empregada como na Fig. 4.20.

A discussão acima da recuperação na modalidade de desempenho não endereçou o importante tema das mensagens de erro. Em geral as mensagens de erro informativas têm que ser fornecidas pelo projetista do compilador.

Recuperação em nível de frase. A recuperação em nível de frase é implementada preenchendo-se as entradas em branco da tabela sintática preditiva com apontadores para rotinas de erro. Essas rotinas podem modificar, inserir ou apagar símbolos da entrada, e emitir as mensagens de erro apropriadas. Podem também remover símbolos da pilha. Uma questão importante é se devemos permitir a alteração dos símbolos da pilha, ou empilhar novos símbolos, uma vez que os passos dados pelo analisador podem não corresponder à derivação de palavra alguma da gramática. Em qualquer caso, devemos estar certos de que haja a possibilidade de um progresso. A verificação de que uma ação de recuperação resulte em um símbolo de entrada, sendo consumido, ou a pilha sendo encurtada se a entrada estiver vazia, é um princípio básico para impedir que um laço infinito ocorra. Vimos exemplos disso em nossos projetos de rotinas de erro.

4.5 ANÁLISE SINTÁTICA *BOTTOM-UP*

Nesta seção, introduzimos o estilo geral de análise sintática *bottom-up*, conhecida como análise de empilhar e reduzir. Uma forma fácil de implementá-la, chamada de análise sintática de precedência de operadores, é apresentada na Seção 4.6. Um método muito mais geral de análise de empilhar e reduzir, chamado de análise LR, é discutido na Seção 4.7. A ferramenta Yacc discutida na Seção 4.9 usa esses métodos de análise LR.

A análise gramatical de empilhar e reduzir tenta construir uma árvore gramatical para uma cadeia de entrada começando pelas folhas (o fundo) e trabalhando a árvore acima em direção à raiz (o topo). Podemos pensar neste processo como o de "reduzir" uma cadeia *w* ao símbolo de partida de uma gramática. A cada passo de redução, uma subcadeia particular que reconhecemos ser lado direito de uma produção é substituída pelo símbolo à esquerda daquela produção, e se a subcadeia for escolhida corretamente a cada passo, uma derivação mais à direita reversa será traçada em ordem inversa.

Exemplo 4.21. Considere a gramática

$$S \rightarrow aABe$$
$$A \rightarrow Abc \mid b$$
$$B \rightarrow d$$

Não-Terminal	Símbolo de Entrada					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	sinc	sinc
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$	sinc		$T \rightarrow FT'$	sinc	sinc
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow$ id	sinc	sinc	$F \rightarrow (E)$	sinc	sinc

Fig. 4.18. Entradas de sincronização adicionadas à tabela sintática da Fig. 4.15.

A operação *poda* pode ser realizada em vários passos:

$$\begin{aligned} &\alpha\beta w \\ &\alpha Aw \\ &\gamma w \\ &\alpha' A' w \\ &S \end{aligned}$$

Podemos esquematizar o processo de poda por uma árvore cuja raiz corresponde à cadeia inicial e cujas folhas correspondem à cadeia que se quer reduzir ao símbolo de partida. Vamos escolher os nós internos e expandi-los por meio de uma produção $A \to \beta$ de modo que os filhos desses nós formem a cadeia $\alpha\beta$. Vemos, na subárvore $\alpha\beta w$, que βw é reconhecível como lado direito de alguma produção. Após escolhermos a subárvore mais à esquerda que reconhece o lado direito de alguma produção, escolhemos substituir a subárvore βw por A, o lado esquerdo da produção $A \to \beta$. Obtemos αAw. Para a substituição de $\alpha\beta$ por γ, o lado esquerdo da produção $B \to \alpha\beta$ obtemos γBw. Pode-se agora substituir toda essa cadeia por S. Consequentemente, estamos de uma seqüência de quatro reduções estamos caminhando na redução oposta de S. Essas reduções de fato, fornecem a seguinte derivação direta à direita na ordem inversa

$$S \Rightarrow \gamma Bw \Rightarrow \alpha\beta w \Rightarrow \alpha Aw \Rightarrow \alpha'A'w \Rightarrow \alpha\beta w$$

Handles

Informalmente, um *handle* é uma subcadeia que reconstrói o lado direito de uma produção cuja redução no não-terminal do lado esquerdo da produção representa um passo no processo de uma derivação mais à direita. Em muitos casos a subcadeia β mais à esquerda que reconhece o lado direito de uma produção $A \to \beta$ não é um *handle*, porque a redução pela produção $A \to \beta$ produz uma cadeia que não pode ser gerada a partir do símbolo de partida. No Exemplo 4.21 se substituirmos b por Ab. Uma segunda cadeia $Abcd$ obtemos a cadeia $aAbcd$, que não pode ser subsequentemente reduzida a S. Por esta razão precisamos fornecer uma definição mais precisa de um *handle*.

Formalmente, um *handle* de uma forma sentencial à direita γ é uma produção $A \to \beta$ e uma posição dentro de γ onde a cadeia β pode ser encontrada e substituída por A de forma a produzir a forma sentencial à direita anterior em uma derivação mais à direita de γ. Ou seja, se $S \Rightarrow^{*}_{rm} \alpha Aw \Rightarrow_{rm} \alpha\beta w$, como $A \to \beta$ na posição seguinte a α, é um *handle* de $\alpha\beta w$. A cadeia à direita do *handle* contém somente símbolos terminais. Note-se que dissemos "um *handle*" em vez de "o *handle*" porque a gramática pode ser ambígua, com mais de uma derivação mais à direita para $\alpha\beta w$. Se a gramática for não ambígua, cada forma sentencial à direita da gramática possui exatamente um *handle*.

No exemplo acima, $aAbcd$ é uma forma sentencial à direita cujo *handle* é $A \to b$ na posição 2. Algumas vezes dizemos que "a cadeia β é um *handle* de $\alpha\beta w$" ou a posição de β e a produção $A \to \beta$ que lhe correspondem estão fortemente claras.

A Fig. 4.20 retrata o *handle* $A \to \beta$ na árvore gramatical da forma sentencial à direita $\alpha\beta w$. O *handle* representa a subárvore completa

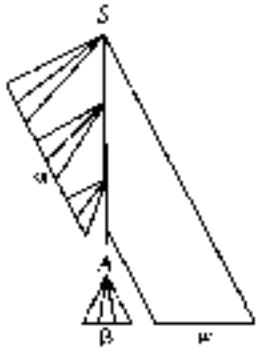


Fig. 4.20. O *handle* $A \to \beta$ na árvore gramatical de $\alpha\beta w$

mais à esquerda consistindo em um nó e todos os seus filhos. Na Fig. 4.20, A é o nó interior mais à esquerda com todos os seus filhos na árvore. A redução de β para A pode ser pensada como a "poda do *handle*", ou seja, a remoção dos filhos de A na árvore gramatical.

Exemplo 4.22. Considere a seguinte gramática

$$\begin{aligned} &(1)\ E \to E + T \mid T \\ &(2)\ T \to T * F \mid F \\ &(3)\ F \to (E) \\ &(4)\ F \to \mathbf{id} \end{aligned} \qquad (4.21)$$

e a derivação mais à direita

$$\begin{aligned} E &\underset{rm}{=} E + T \\ &\underset{rm}{\Rightarrow} E + T * F \\ &\underset{rm}{\Rightarrow} E + T * \mathbf{id}_5 \\ &\underset{rm}{\Rightarrow} E + F * \mathbf{id}_5 \\ &\underset{rm}{\Rightarrow} E + \mathbf{id}_3 * \mathbf{id}_5 \\ &\underset{rm}{\Rightarrow} T + \mathbf{id}_3 * \mathbf{id}_5 \\ &\underset{rm}{\Rightarrow} F + \mathbf{id}_3 * \mathbf{id}_5 \\ &\underset{rm}{\Rightarrow} \mathbf{id}_1 + \mathbf{id}_3 * \mathbf{id}_5 \end{aligned}$$

Sublinhamos os ids para uma conveniência de notação e subscritos para distinguir cada um deles. Cada forma sentencial à direita acima é um *handle* da forma sentencial à direita seguinte. Por exemplo, $\mathbf{id}_1 + \mathbf{id}_3 * \mathbf{id}_5$ é um *handle* da forma sentencial à direita $F + \mathbf{id}_3 * \mathbf{id}_5$, porque $\mathbf{id}_1$ é o lado direito da produção $E \to \mathbf{id}$ e a substituição de $\mathbf{id}_1$ por F produz a forma sentencial à direita $F + \mathbf{id}_3 * \mathbf{id}_5$. Note-se que a cadeia à direita de um *handle* contém somente símbolos terminais.

Como a gramática (4.21) é ambígua, existe uma outra derivação mais à direita para a mesma cadeia:

$$\begin{aligned} E &\underset{rm}{=} E * F \\ &\underset{rm}{\Rightarrow} E * \mathbf{id}_5 \\ &\underset{rm}{\Rightarrow} E + F * \mathbf{id}_5 \\ &\underset{rm}{\Rightarrow} E + \mathbf{id}_3 * \mathbf{id}_5 \\ &\underset{rm}{\Rightarrow} T + \mathbf{id}_3 * \mathbf{id}_5 \\ &\underset{rm}{\Rightarrow} F + \mathbf{id}_3 * \mathbf{id}_5 \\ &\underset{rm}{\Rightarrow} \mathbf{id}_1 + \mathbf{id}_3 * \mathbf{id}_5 \end{aligned}$$

Cerca de cada uma das formas sentenciais à direita $E + E * \mathbf{id}$. Nesta derivação, $E + E$ é um *handle* de $E + E * \mathbf{id}_5$, enquanto que $\mathbf{id}_5$ é um *handle* desta mesma forma sentencial à direita de acordo com a derivação anterior.

As duas derivações mais à direita neste exemplo são análogas às duas derivações mais à esquerda no Exemplo 4.6. A primeira derivação corresponde à "uma maior precedência de que" enquanto que a segunda dá "a mesma precedência".

A Poda do *Handle*

Uma derivação mais à direita na ordem inversa pode ser obtida "podando os *handles*". Ou seja, começa-se pela primeira cadeia de terminais w que desejamos reconhecer. Se w for uma sentença da gramática em questão, então $w = \gamma_n$, onde γ_n é a enésima forma sentencial à direita de alguma derivação mais à direita até de completa da

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

Para reconstruir esta derivação na ordem inversa, localizamos o *handle* β_n em γ_n e substituímos β_n pelo lado direito de alguma produção $A_n \to \beta_n$, de modo a obtermos a (n-1)ésima forma sentencial à direita γ_{n-1}. Note-se que ainda não sabemos como os *handles* são encontrados, mas em breve veremos os métodos para tal.

Repetimos, em seguida, esse processo. Isto é, localizamos o *handle* β_{n-1} em γ_{n-1} e o reduzimos de forma a obter a forma sentencial à direita γ_{n-2}. Continuando esse processo, produzimos uma forma sentencial à direita consistindo somente no símbolo de partida S e en-

à operação a ser executada e o terminal com o símbolo da análise sintática. O reverso da sequência de produções usadas nas reduções é uma derivação mais à direita para a cadeia de entrada.

Exemplo 4.23. Considere a gramática do Exemplo 4.22 e a cadeia de entrada **id** * **id** + **id**. A sequência de substituições mostrada na Fig. 4.21 reduz **id**$_1$ * **id**$_2$ + **id**$_3$ ao símbolo de partida E. O leitor deverá observar que a sequência de formas sentenciais obtida neste exemplo é exatamente o reverso da sequência da primeira derivação mais à direita no Exemplo 4.22. □

FORMA SENTENCIAL	LEXEMA	PRODUÇÃO REDUTORA
$\mathbf{id}_1 * \mathbf{id}_2 + \mathbf{id}_3$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2 + \mathbf{id}_3$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * \mathbf{id}_2 + \mathbf{id}_3$	$\mathbf{id}_3$	$F \to \mathbf{id}$
$T * F + \mathbf{id}_3$		$T \to T * F$
$T + \mathbf{id}_3$		$T \to F$
$E + \mathbf{id}_3$		$E \to E + T$
$E + F$		$F \to \mathbf{id}$
$E + T$		$T \to F$
E		$E \to E + T$

Fig. 4.21. Reduções realizadas pelo analisador sintático de empilhar e reduzir.

Implementação de Pilha da Análise Sintática de Empilhar e Reduzir

Existem dois problemas que precisam ser resolvidos se quisermos dispor a análise sintática de empilhar e reduzir através da pilha de *lexemas*. O primeiro é de localizar a subcadeia a ser reduzida em uma forma sentencial. O segundo é de determinar que produção escolher no caso de existir mais de uma produção com aquela subcadeia no lado direito. Antes de enfrentarmos essas questões, vamos considerar os tipos de estruturas de dados usados em um analisador sintático de empilhar e reduzir.

Uma implementação eficiente usa uma pilha para armazenar símbolos gramaticais e um *buffer* de entrada para a cadeia a ser decomposta. Usamos $\$$ para marcar o fundo da pilha e também o final à direita da cadeia de entrada inicialmente. A pilha está vazia e a cadeia de entrada contém a cadeia $w\$$, como segue:

PILHA	ENTRADA
$\$$	$w\$$

O analisador sintático opera empilhando zero ou mais símbolos na pilha (até que um *handle* β surja no topo da pilha). Reduz então β para o lado esquerdo da produção apropriada. Repete este ciclo até que seja detectado um erro ou até que a pilha contenha o símbolo de partida e a entrada esteja vazia:

PILHA	ENTRADA
$\$S$	$\$$

Após entrar nesta configuração, para e anuncia término com sucesso da análise sintática.

Exemplo 4.24. Vamos rastrear as ações que um analisador sintático de empilhar e reduzir realizaria ao decompor a cadeia de entrada **id**$_1$ * **id**$_2$ + **id**$_3$, de acordo com a gramática (4.1–3), usando a primeira derivação do Exemplo 4.22. A sequência é mostrada na Fig. 4.22. Note que, em cada configuração, há duas derivações mais à direita para a entrada, e existe uma certa sequência de ações que um analisador sintático de empilhar e reduzir poderia seguir.

PILHA	ENTRADA	AÇÃO
(1) $\$$	$\mathbf{id}_1 * \mathbf{id}_2 + \mathbf{id}_3\$$	empilhar
(2) $\$\mathbf{id}_1$	$* \mathbf{id}_2 + \mathbf{id}_3\$$	reduzir por $F \to \mathbf{id}$
(3) $\$F$	$* \mathbf{id}_2 + \mathbf{id}_3\$$	empilhar
(4) $\$F*$	$\mathbf{id}_2 + \mathbf{id}_3\$$	empilhar
(5) $\$F * \mathbf{id}_2$	$+ \mathbf{id}_3\$$	reduzir por $F \to \mathbf{id}$
(6) $\$F * F$	$+ \mathbf{id}_3\$$	reduzir por $T \to F$
(7) $\$T$	$+ \mathbf{id}_3\$$	empilhar
(8) $\$T+$	$\mathbf{id}_3\$$	empilhar
(9) $\$T + \mathbf{id}_3$	$\$$	reduzir por $F \to \mathbf{id}$
(10) $\$T + F$	$\$$	reduzir por $T \to T * F$
(11) $\$T$	$\$$	reduzir por $E \to E + T$
(12) $\$E$	$\$$	aceitar

Fig. 4.22. Configurações de um analisador sintático de empilhar e reduzir para a entrada $\mathbf{id}_1 * \mathbf{id}_2 + \mathbf{id}_3$.

Conquanto as operações principais de um analisador sintático de empilhar e reduzir sejam obviamente empilhar e reduzir, existem de fato quatro ações possíveis que o mesmo pode realizar: (1) empilhar, (2) reduzir, (3) aceitar e (4) erro.

1. Numa ação de *empilhar*, o próximo símbolo de entrada é empurrado no topo da pilha.
2. Numa ação de *redução*, o analisador sabe que o final (à direita) de um *handle* está no topo da pilha. Precisa, então, localizar o início à esquerda do *handle* dentro da pilha e decidir qual não terminal substituir o *handle*.
3. Numa ação de *aceitar*, o analisador anuncia término com sucesso da operação de decomposição.
4. Numa ação de *erro*, o analisador descobre que um erro sintático ocorreu e chama uma rotina de recuperação de erros.

Existe um fato importante que justifica o uso de uma pilha numa análise sintática de empilhar e reduzir: o *handle* irá sempre aparecer no topo da pilha, nunca dentro dela. Este fato será bem fácil de ser considerado após as possíveis formas que dois passos numa derivação mais à direita podem ter. Dados dois passos quaisquer de uma derivação mais à direita, podemos ter:

(1) $S \underset{rm}{\Rightarrow} \alpha A z \underset{rm}{\Rightarrow} \alpha\beta y z$

(2) $S \underset{rm}{\Rightarrow} \alpha B x A z \underset{rm}{\Rightarrow} \alpha B x \gamma y z$

No caso (1), A é substituído por βy e, em seguida, B é substituído mais à direita naquela forma sentencial. É substituído por γ. No caso (2), A é diretamente substituído em primeiro lugar, mas desta vez a cadeia à direita de A contém somente terminais. O próximo não-terminal mais à direita B estará, portanto, em algum lugar à esquerda de A.

Vamos considerar o caso (1) em contrário, onde um analisador de empilhar e reduzir acabou de atingir a configuração:

PILHA	ENTRADA
$\$\alpha\beta y$	$z\$$

O analisador agora executa *redução* para B e atinge a configuração:

PILHA	ENTRADA
$\$\alpha B$	$yz\$$

Como B é o não-terminal mais à direita em $\alpha B y z$, é fácil localizar o *handle* de $\alpha B y$ e não precisa procurar dentro da pilha. Por conseguinte, o analisador pode empilhar a cadeia y de modo a atingir a configuração:

PILHA	ENTRADA
$\$\alpha\beta y$	$z\$$

na qual βy é o *handle*, o qual se vê reduzido a A.

Uma vez que a distinção entre símbolos terminais e não-terminais é crucial para o funcionamento correto deste método de análise sintática, A não pode ser um não-terminal nulo.

No caso (2) a configuração

PILHA	ENTRADA
$S a T$	$c \$$

deixa c no topo da pilha. Após reduzir por $B \to c$, o analisador sintático empilha c e B de modo a obter a configuração

PILHA	ENTRADA
$S a T B$	$c \$$

Esquematize o passo (3).

Imaginamos o caso após realizar uma redução, o analisador sintático precisa empilhar zero ou mais símbolos de modo a ter o lado direito desejado no topo da pilha. Jamais teve que olhar no fundo da pilha para reconhecer qualquer aspecto da pilha. Portanto, podemos substituir a pilha por uma estrutura de dados pouco diferente com o objetivo de representar um subconjunto viável de símbolos que podem ser lidos e escolhidos após a serem realizadas todas as reduções pelo analisador sintático de empilhar e reduzir. Isto torna correta a redução seguinte. A precedência de operadores é outro exemplo de um método sintático que lida com uma pilha de símbolos gramaticais, mas não examina mais do que um número limitado de símbolos do topo da pilha.

Prefixos Viáveis

Especificamos que uma característica essencial de qualquer algoritmo que possa figurar na pilha de um analisador sintático de empilhar e reduzir é que ele deve ser um prefixo de uma forma sentencial. Lembre-se que uma forma sentencial é uma derivação de uma única produção, ou seja, $S \Rightarrow^* \alpha$. Em geral, uma forma sentencial pode conter tanto terminais quanto não terminais e, de fato, a cadeia vazia é uma forma sentencial trivial. Porém, se sempre mantivermos símbolos gramaticais válidos em nossa pilha, então a entrada que ainda não foi lida deve ser uma cadeia de símbolos terminais que pode ser derivada a partir do símbolo à direita da pilha. Por exemplo, se a pilha contém $A_1 A_2 \cdots A_m$ (com A_m no topo) e a entrada restante é $a_i a_{i+1} \cdots a_n \$$, então existe uma derivação

$$S \Rightarrow^* A_1 A_2 \cdots A_m a_i a_{i+1} \cdots a_n \$$$

Em termos de árvores de derivação, isto significa que podemos construir uma árvore de derivação para a entrada $a_i a_{i+1} \cdots a_n \$$ cujas folhas, lidas da esquerda para a direita, são $A_1 A_2 \cdots A_m a_i a_{i+1} \cdots a_n \$$. Não podemos garantir que a pilha $A_1 A_2 \cdots A_m$ seja exatamente o que o analisador sintático de empilhar e reduzir construiu até o momento, pois ele pode ter reduzido uma cadeia β para A_m e, em seguida, empilhado A_m e A_{m+1} para obter a pilha $A_1 A_2 \cdots A_m A_{m+1}$. Porém, $A_1 A_2 \cdots A_m$ nunca vai ser mais do que a pilha construída pelo analisador sintático e nunca vai ser menos do que a pilha construída pelo analisador sintático após a redução de β para A_m.

Definimos um **prefixo viável** como qualquer prefixo de uma forma sentencial γ que não se estende além do final do lado direito mais à direita de γ. Por exemplo, se $S \Rightarrow^* \alpha A w \Rightarrow \alpha \beta w$, então $\alpha \beta$ é um prefixo viável. Se $\alpha \beta$ for uma forma sentencial, então $\alpha \beta$ é um prefixo viável. Se $\alpha \beta$ não for uma forma sentencial, então $\alpha \beta$ não é um prefixo viável. Portanto, a cadeia vazia é sempre um prefixo viável. Também podemos mostrar que todos os prefixos de $\alpha \beta$ são prefixos viáveis. Por fim, não há nada além de $\alpha \beta$ que seja um prefixo viável. Para ver por quê, observe que, por definição, $\alpha \beta$ é o lado direito mais à direita de qualquer forma sentencial que o contenha. Não podemos ir além do final de $\alpha \beta$ sem ir além de qualquer forma sentencial que o contenha.

Em resumo, o conjunto de prefixos viáveis é exatamente o conjunto de prefixos de formas sentenciais direitas. Uma vantagem de saber que a pilha sempre contém um prefixo viável é que podemos adiar a decisão sobre qual ação tomar em relação a uma cadeia β no topo da pilha até que tenhamos empilhado mais símbolos gramaticais sobre β. Em outras palavras, nunca saberemos com certeza se uma cadeia β no topo da pilha é o lado direito de uma produção até que tenhamos visto os símbolos de entrada que seguem β. Por exemplo, não podemos saber se β deve ser reduzido para A ou se β deve ser empilhado até que tenhamos visto a.

Conflitos Durante a Análise Sintática de Empilhar e Reduzir

Existem gramáticas livres de contexto para as quais o método de empilhar e reduzir não pode ser usado. Cada analisador de empilhar e reduzir para uma tal gramática pode atingir uma configuração na qual, mesmo conhecendo todo o conteúdo da pilha e o próximo símbolo de entrada, não pode decidir entre empilhar ou reduzir (um **conflito empilhar/reduzir**) ou não pode decidir qual das duas reduções alternativas deve realizar (um **conflito redução/redução**). Vamos agora apresentar duas construções sintáticas que são comuns em gramáticas para linguagens de programação. Descobriremos que as gramáticas que usam essas construções não estão na classe de gramáticas LR(1) definida na Seção 4.3, ou seja, não são gramáticas LR(1). Em seguida, vamos ver duas formas de lidar com o conflito. Uma delas é reescrever a gramática de modo que ela seja LR(1) e use a mesma linguagem. A outra é usar uma regra especial durante a análise sintática de empilhar e reduzir.

Exemplo 4.25. Uma gramática ambígua para uma construção `if-then-else` é

$$\begin{aligned} stmt &\to \textbf{if } expr \textbf{ then } stmt \\ &\mid \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ &\mid outro \end{aligned}$$

Se tivermos um analisador sintático de empilhar e reduzir na configuração

PILHA	ENTRADA
$\cdots$ **if** $expr$ **then** $stmt$	**else** $\cdots \$$

não podemos dizer se **if** $expr$ **then** $stmt$ é o lado direito sem importar o que aparece abaixo do topo da pilha. Após uma sentença empilhada de `if` dependendo do que aparece abaixo do topo da pilha, a entrada `else` pode ser empilhada, pois `if expr then stmt` pode ser reduzido para `stmt`, ou pode ser necessário empilhar `else` e mais alguns símbolos para completar a alternativa `if expr then stmt else stmt`. Consequentemente, nos próximos dois exemplos descrevemos duas maneiras de reescrever a gramática para ser LR(1). Mais geralmente, nenhuma gramática ambígua pode ser LR(1), e frequentemente esse caso pode ser LR(1) para qualquer que seja o valor de k.

Devemos mencionar, entretanto, que a análise sintática de empilhar e reduzir pode ser facilmente adaptada para decompor muitas gramáticas ambíguas, tais como a gramática acima. Quando construímos um analisador para uma gramática com duas produções como essa, podemos escolher empilhar em `else`, ou reduzir `if expr then stmt`. Se resolvermos o conflito em favor de empilhar, o analisador sintático se comporta da maneira natural. Discutiremos essas técnicas para gramáticas ambíguas na Seção 4.8.

Outra causa comum da não aplicabilidade à análise sintática de empilhar e reduzir ocorre quando sabemos que um lado direito é o correto ao ver o símbolo de entrada, mas não suficiente para determinar qual produção deve ser usada numa redução. O próximo exemplo ilustra esta situação.

Exemplo 4.26. Suponha que temos um analisador sintático que reconhece `id` para todos os identificadores independentemente de seu tipo. Suponha também que nessa linguagem invoque procedimentos ou funções com parâmetros entre parênteses e que os argumentos sejam referências a variáveis através de nossa sintaxe usual. Como a distinção entre funções e procedimentos não afeta a análise sintática, podemos usar produções distintas para gerar as listas de parâmetros atuais de funções e procedimentos. Nossa gramática poderia ter, por consequência, produções e regras de tradução como

(1)	$stmt \to \textbf{id}$ ($lista\_de\_parametros$)	
(2)	$stmt \to expr = expr$	
(3)	$lista\_de\_parametros \to lista\_de\_parametros$, $parametro$	
(4)	$lista\_de\_parametros \to parametro$	
(5)	$parametro \to \textbf{id}$	
(6)	$expr \to \textbf{id}$ ($lista\_de\_expr$)	
(7)	$expr \to \textbf{id}$	
(8)	$lista\_expr \to lista\_expr$, $expr$	
(9)	$lista\_expr \to expr$	

Um comando, começando por `id(id, id)`, aparecerá para o analisador sintático como o fluxo de tokens `id(id, id)`. Após empilhar os três primeiros tokens, um analisador de empilhar e reduzir estaria na configuração

PILHA	ENTRADA
$\cdots$ **id** (**id**	, **id**) $\cdots$

Em seguida, queremos reduzir `id` para $parametro$ (usando a produção (5)), mas também gostaríamos de empilhar , e mais um símbolo para preparar a redução para $lista\_de\_parametros$. Infelizmente, não podemos saber se `id` é um nome de procedimento ou um nome de variável até muito mais tarde. Em particular, não podemos saber se `id` é um nome de procedimento até que tenhamos visto o `:=` após o `)`. Até lá, não há nada em nossa pilha que nos diga se o `id` é um nome de procedimento ou um nome de variável, pois as informações na tabela de símbolos, obtidas a partir da declaração de `id`, precisam ser usadas.

Uma solução é modificar a gramática para que ela não faça distinção entre os diferentes tipos de identificadores. Podemos ter uma única produção para substituir as produções (5) e (7), tais como $parametro \to \textbf{id}$ e $expr \to \textbf{id}$. O analisador sintático, então, não precisará saber o tipo de `id` que está vendo. A verificação de tipos ainda deverá ser feita mais tarde, mas a análise sintática pode ser guiada pela sintaxe. Outra solução é fazer com que o analisador sintático consulte a tabela de símbolos, à procura do tipo de `id`, e use essa informação ao decidir qual ação tomar. Em geral, essa solução é preferível, pois sabemos que a análise sintática pode ser guiada pela sintaxe, mas a análise semântica é necessária antes que possamos ter certeza de que um programa pode ser executado.

Se reescrevermos a gramática, o analisador sintático de empilhar e reduzir estaria na configuração

PILHA	ENTRADA
$\cdots$ **procid** (**id**	, **id**) $\cdots$

em uma configuração imediatamente anterior. Se empilharmos, ele terá a chance de reconhecer a lista de parâmetros através da produção (3), ou reconhecer um parâmetro através da produção (5). Observe como o terceiro símbolo a partir do topo da pilha de cada

termina a redução a ser realizada, ainda que não exista ambiguidade na redução. A análise sintática de empilhar e reduzir pode ser utilizada de forma muito abrangente para gramáticas decomposições [1

4.6 ANÁLISE SINTÁTICA DE PRECEDÊNCIA DE OPERADORES

A mais ampla classe de gramáticas para a qual os analisadores sintáticos de empilhar e reduzir podem ser construídos com sucesso — as gramáticas LR —, será discutida na Seção 4.7. Entretanto, para uma pequena porém importante classe de gramáticas, podemos facilmente construir à mão analisadores sintáticos de empilhar e reduzir. Essas gramáticas possuem a propriedade (dentre outras exigências) de que nenhum lado direito de produção seja ε ou contenha dois não-terminais adjacentes. Uma gramática com a última propriedade é chamada de uma *gramática de operadores*.

Exemplo 4.27. A seguinte gramática para expressões

$$E \to E A E \mid (E) \mid -E \mid \mathbf{id}$$
$$A \to + \mid - \mid * \mid / \mid \uparrow$$

não é uma gramática de operadores porque o lado direito EAE possui dois não-terminais consecutivos. Entretanto, se substituirmos A por cada uma de suas alternativas, obtemos a seguinte gramática de operadores:

$$E \to E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid -E \mid \mathbf{id}$$

Descreveremos agora uma técnica de análise sintática (ad hoc) de uma determinada variedade de analisadores sintáticos de precedência de operadores. Historicamente, esses analisadores foram os primeiros a serem implementados sobre computadores reais. Uma gramática substancialmente mais ampla pode ser analisada com a técnica de precedência de operadores a partir de uma gramática, porém ignoraremos esta última, pois não é possível operar com facilidade sobre os atributos associados aos não-terminais.

Uma propriedade geral de analisadores sintáticos de precedência de operadores possui várias vantagens. Por exemplo, é difícil tratar as formas comuns de sinal de menos, que possuem duas diferentes precedências (dependendo de ser unário ou binário). Selecionado, para cada uma delas, o símbolo terminal apropriado a ser retornado pelo analisador léxico, o analisador sintático de precedência de operadores é muito fácil de ser construído. Infelizmente, somente uma pequena classe de gramáticas pode ser decomposta usando as técnicas de precedência de operadores.

Apesar de tudo, devido à sua implicidade, razoáveis compiladores podem ser construídos usando técnicas de análise sintática de precedência de operadores para linguagens suficientemente interessantes. Frequentemente, esses analisadores são de precedência recuperável, como descrito na Seção 4.4, para mensagens de erro e outras construções de alto nível. Analisadores sintáticos de precedência de operadores também são bastante usados para linguagens internas.

Na análise sintática de precedência de operadores, definimos três relações *precedência de operadores* disjuntas $<\cdot$, $\doteq$ e $\cdot>$ entre certos pares de terminais. Essas relações direcionam a seleção de *frases* a serem substituídas durante a redução. Se a e b são terminais, então exatamente uma das seguintes relações é válida:

Relação	Significado
$a <\cdot b$	a "cede precedência para" b
$a \doteq b$	a possui a mesma precedência que b
$a \cdot> b$	a "toma precedência sobre" b

Devemos prevenir o leitor de que, enquanto essas relações possam parecer similares às relações aritméticas "menor do que", "igual a" e "maior do que", as relações de precedência possuem propriedades bastante diferentes. Por exemplo, poderíamos ter $a <\cdot b$ e $a \cdot> b$ para uma dada linguagem, ou poderíamos ter nenhuma relação válida para a dupla de terminais (a, b).

Existem várias formas comuns de determinar quais relações de precedência devem ser válidas para um par de terminais. O primeiro método que discutiremos é intuitivo e está baseado nas noções tradicionais de associatividade e precedência de operadores. Por exemplo, se $*$ deve ter uma precedência maior do que $+$, escrevemos $+ <\cdot *$ e $* \cdot> +$. Esse método será descrito em detalhes após algumas gramáticas simples terem sido estudadas. Entretanto, a técnica para escrever e analisar precedências de operadores também pode ser aplicada a gramáticas para as quais as noções tradicionais de precedência de operadores não são apropriadas.

Um segundo método de selecionar as relações de precedência de operadores consiste em primeiro construir, para a linguagem, uma gramática não ambígua que reflita a associatividade e precedências corretas em suas árvores gramaticais. Esta gramática é diferente das expressões usadas na discussão das expressões na Seção 2.2 porque ela não deve ser ambígua. Para as outras fontes comuns de ambiguidade, como a regra **else**-vazio da gramática (4.9) ou o modelo (4.11), também é possível construir uma gramática não ambígua para se construir relações de precedência de operadores a partir dela mesma. Essas construções podem ser difíceis e podem demorar muito. Uma linguagem que não é gerada pela gramática, mas, sim, o tipo de listas de expressões aritméticas, pode ser problemático na prática. Nós iremos discutir esta construção, a qual se deve a Aho e Ullman [1972b].

Usando Relações de Precedência de Operadores

O objetivo das relações de precedência é delimitar o *handle* de uma forma sentencial. Diremos com "associando o handle à esquerda" ligando por enquanto o modelo $\cdot>$ marcando o término à direita. Para termos isso possível, suponhamos que a forma sentencial à direita de uma derivação canônica de operadores não tenha dois não-terminais adjacentes nos lados direitos de produções implica igualmente que nenhuma forma sentencial à direita tenha dois não-terminais adjacentes. Portanto, podemos escrever a forma sentencial à direita como $\beta_0 \beta_1 \cdots \beta_n$, onde cada β_i é ou a cadeia vazia ou um único não-terminal e cada β_i é um único terminal.

Suponhamos que entre cada par de terminais adjacentes β_i e β_{i+1} exatamente uma das três relações $<\cdot$, $\doteq$ ou $\cdot>$ seja válida. Adicionemos também $\$$ nos dois lados da cadeia e definamos que $\$ <\cdot a$ e $a \cdot> \$$ para todos os terminais a. Suponhamos agora que tentamos remover todos os não-terminais da cadeia e colocarmos a relação correta $<\cdot$, $\doteq$ ou $\cdot>$ entre cada par de terminais e entre os terminais das extremidades e os erros. Desse modo, as frases na esquerda e à direita da cadeia. Por exemplo, suponhamos que inicialmente tenhamos a forma sentencial $\mathbf{id} + \mathbf{id} * \mathbf{id}$. As relações de precedência a seguir aquelas listadas na Fig. 4.23. Essas relações são algumas daquelas que escolheríamos de forma a descrever uma expressão de acordo com a gramática (4.7).

	id	+	*	$
id		$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	

Fig. 4.23. Relações de precedência de operadores.

Consequentemente, a cadeia com as relações de precedência inseridas é

$$\$ <\cdot \mathbf{id} \cdot> + <\cdot \mathbf{id} \cdot> * <\cdot \mathbf{id} \cdot> \$ \qquad (4.10)$$

Por exemplo, $<\cdot$ é inserida entre $\$$ e $+$ pois $+$ está à esquerda e id, uma vez que $+$ está contida na tabela na coluna id. O handle pode ser encontrado pelo seguinte processo:

1. Percorra a cadeia a partir da extremidade da esquerda até que a primeira $\cdot>$ seja encontrada. Em (4.10), essa é a primeira $\cdot>$ entre a primeira id e $+$.

2. Empilhar o símbolo de entrada para a esquerda, possivelmente sobre uma cadeia de relações $\doteq$ até que o símbolo encontrado tenha precedência mais alta do que o símbolo de entrada.
3. O marcador contém todos os símbolos à esquerda do primeiro $\lessdot$ ou $\doteq$ encontrado no passo (2). Juntamente com qualquer número de não-terminais adjacentes não relacionados por precedência. (A inclusão de não-terminais adjacentes não é necessária para que dois não-terminais adjacentes não figurem na forma sentencial, mas é conveniente. Em (4.36), o marcador é apenas **id**.

Se estivermos fazendo a tradução, a ação (4.1.6) relacionada com **id** pode ser **id**. Neste ponto, temos a forma sentencial E **id** $+$ **id**. Após reduzirmos os dois **id** a restantes a E, obtemos a forma sentencial $E + E$. Considerando agora a cadeia $\$ + \$$, símbolo pela remoção dos não-terminais, lembrando as relações de precedência, obtemos

$$\$ \lessdot + \gtrdot \$$$

indicando que a extremidade à esquerda do *manúsculo* está entre o $+$ e o $\$$ e a extremidade direita entre os dois $\$$. As relações de precedência indicam que, na forma sentencial reduzida $E + E$, o *manúsculo* é $E + E$. Sabemos que os E não pertencem ao *manúsculo*.

Uma vez que os não-terminais não influenciam a decomposição, não precisamos nos preocupar com a distinção entre os operadores. Um "analisador sintático de empilhar e reduzir" pode ser montado em pilha de notação sintático de empilhar e reduzir a fim de ordenar os marcadores de ação para os valores de atributos.

Pode parecer, a partir da discussão acima, que toda a forma sentencial à direita precisa ser reconstruída a cada passo para se encontrar o *manúsculo*. Tal não é o caso se usarmos uma pilha para armazenar os símbolos de entrada já encaminhados e as relações de precedência lembradas para guiar ações de uma análise sintática de empilhar e reduzir. Se uma das relações de precedência $\lessdot$ ou $\doteq$ ocorre entre o símbolo terminal no topo da pilha e o próximo símbolo de entrada, a análise empilha, ainda não se sabendo a extremidade direita do *manúsculo*. Se a relação $\gtrdot$ for encontrada, uma redução é pedida. A estrutura da análise encontra a extremidade direita do *manúsculo* e as relações de precedência podem ser usadas para encontrar a extremidade esquerda do *manúsculo* dentro da pilha.

Se nenhuma relação de precedência ocorrer entre um par de símbolos (indicada por uma entrada em branco na Fig. 4.23), então uma rotina de tratamento de erro sintático foi detectada e uma rotina de recuperação de erros deve ser invocada, como será descrito mais adiante nesta seção. As ações de análise podem ser formalizadas pelo algoritmo que se segue.

Algoritmo 4.5. Algoritmo para a análise sintática de precedência de operadores.

```
(1)  Fazer ap apontar para o primeiro símbolo de w$
(2)  repetir para sempre
(3)     se s estiver no topo da pilha e ap apontar para $ então
(4)        retornar
         senão início
(5)        seja a o símbolo apontado por ap na entrada;
           e seja b o símbolo mais alto da pilha
(6)        se a ≐ b ou a ⋖ b então início
(7)           empilhar a;
(8)           avançar ap de forma a apontar para o próximo símbolo
              da entrada
           fim;
(9)        senão se a ⋗ b então reduzir
(10)          repetir
(11)             remover o topo da pilha
(12)          até que o terminal mais recentemente removido
                 tenha relação ⋖ com o terminal mais recentemente removido
(13)       senão erro()
         fim
```

Fig. 4.24. Algoritmo de análise sintática de precedência de operadores.

Entrada. Uma cadeia w com marcadores de relações de precedência.

Saída. Se w for bem formada, uma representação de árvore gramatical com w como fronteira, produzida de algum modo. Caso contrário, uma indicação de erro.

Método. Inicialmente a pilha contém s, o símbolo de entrada contém w\$. Para descompilar, executamos o programa da Fig. 4.24.

Relações de Precedência de Operadores a partir da Associatividade e Precedência

Estamos sempre prontos para criar relações de precedência de operadores a qualquer ponto em que as regras ambíguas e especiais que algoritmo de análise sintática de precedência de operadores não reconhecerá corretamente quando guiado por elas. Para uma linguagem de expressões aritméticas, tal como a gerada pela gramática (4.17), podemos usar o seguinte procedimento heurístico para produzir o conjunto apropriado de relações de precedência. Note-se que a gramática (4.17) é ambígua e que as formas sentenciais a ela associadas poderiam conter vários *manúsculos*. Nossas regras são projetadas para selecionarmos os *manúsculos* corretos, que refletem um conjunto de regras de precedência e associatividade para operadores binários.

1. Se o operador θ_1 possui maior precedência do que o operador θ_2, fazer $\theta_1 \gtrdot \theta_2$ e $\theta_2 \lessdot \theta_1$. Por exemplo, se $*$ tiver maior precedência do que $+$, fazer $* \gtrdot +$ e $+ \lessdot *$. Essas relações asseguram que, numa expressão da forma $A + B * C + D$, o manúsculo que será reduzido primeiro é $B * C$.
2. Se θ_1 e θ_2 são operadores de igual precedência (podem ser um mesmo operador), fazer $\theta_1 \gtrdot \theta_2$ e $\theta_2 \gtrdot \theta_1$ se os operadores forem associativos à esquerda, ou fazer $\theta_1 \lessdot \theta_2$ e $\theta_2 \lessdot \theta_1$ se forem associativos à direita. Por exemplo, se $+$ e $-$ forem associativos à esquerda e não fazer $+ \lessdot -$ e $- \lessdot +$. Se $\uparrow$ for associativo à direita, então fazer $\uparrow \lessdot \uparrow$. Essas relações asseguram que $E - E - E$ será tratado como $(E - E) - E$ selecionando e que $E \uparrow E \uparrow E$ terá a ação $E \uparrow (E \uparrow E)$ selecionada.
3. Fazer $\theta \lessdot$ **id**, **id** $\gtrdot \theta$, $\theta \lessdot (\,$, $) \gtrdot \theta$, $\theta \gtrdot)$, $\$ \lessdot \theta$ e $\theta \gtrdot \$$ para todos os operadores θ. Fazer também

$$(\doteq) \qquad (\lessdot (\qquad (\lessdot \textbf{id}$$
$$\$ \lessdot (\qquad \textbf{id} \gtrdot) \qquad) \gtrdot)$$
$$\$ \lessdot \textbf{id} \qquad \textbf{id} \gtrdot \$ \qquad) \gtrdot \$$$

Essas regras asseguram que tanto **id** quanto $(\,(\,)\,)$ serão reduzidos a E. Igualmente, $\$$ será tratado como marcador do extremidade esquerda quanto da direita, fazendo com que os *manúsculos* sejam encontrados entre os $\$$ sempre que possível.

Exemplo 4.28. A Fig. 4.25 contém relações de precedência de operadores para a gramática (4.17), assumindo que

1. $\uparrow$ possua o maior precedência e seja associativo à direita,

	+	−	*	/	↑	**id**	(	)	$
+	⋗	⋗	⋖	⋖	⋖	⋖	⋖	⋗	⋗
−	⋗	⋗	⋖	⋖	⋖	⋖	⋖	⋗	⋗
*	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
/	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
↑	⋗	⋗	⋗	⋗	⋖	⋖	⋖	⋗	⋗
id	⋗	⋗	⋗	⋗	⋗			⋗	⋗
(	⋖	⋖	⋖	⋖	⋖	⋖	⋖	≐	
)	⋗	⋗	⋗	⋗	⋗			⋗	⋗
$	⋖	⋖	⋖	⋖	⋖	⋖	⋖		

Fig. 4.25. Relações de precedência de operadores.

2. Se θ for um operador à esquerda e θ_1 tiver precedência mais alta que θ, então θ_1 também terá precedência mais alta que θ.
3. Se θ e θ_1 forem operadores à esquerda com a mesma precedência, então θ terá precedência mais alta que θ_1 se θ aparecer antes de θ_1 na cadeia de entrada.

Blocos de parênteses aninhados devem ser tratados de forma independente. Dentro de um par de parênteses, a precedência dos operadores é determinada pelas regras anteriores, ignorando todos os operadores fora do par. Assim, podemos tratar cada par de parênteses como se fosse um operador à esquerda com precedência mais alta que qualquer operador que o envolva. Podemos atribuir funções de precedência a esses parênteses aumentando as funções de precedência de todos os operadores de entrada por um valor constante, se necessário, para garantir que nenhum operador tenha uma precedência menor que a dos parênteses. Por exemplo, se $f(\theta) = 2$ e $g(\theta) = 1$ para todos os operadores θ da Fig. 4.19, podemos atribuir $f(\text{id}) = 4$, $g(\text{id}) = 3$, $f(() = g(() = 0$ e $f()) = g()) = 5$. Vírgula para os operadores binários e vírgula final podem ser tratadas de forma semelhante à dos parênteses. Uma exceção é a vírgula que separa os argumentos de uma função, que será discutida na Seção 4.4.3.

Funções de Precedência

Em vez de armazenar tabelas de relações de precedência, muitos compiladores de operadores usam funções que mapeiam símbolos para números inteiros, de forma que a precedência relativa de dois símbolos possa ser determinada pela comparação de seus números inteiros. Em muitos casos, as funções de precedência ocupam menos espaço que a tabela de relações de precedência, mesmo quando a tabela é codificada de forma compacta. Mesmo quando a tabela não puder ser codificada por funções de precedência, o fato de que operadores similares tenham precedências semelhantes pode levar a uma tabela compacta.

Algoritmo 4.8. Construção de funções de precedência.

Entrada. Uma matriz de precedência de operadores.

Saída. Funções de precedência, ou uma indicação de que elas não existem.

Método

1. Crie os símbolos f_a e g_a para cada a que seja um terminal ou $\$$.
2. Particione os símbolos criados em tantos grupos quantos sejam possíveis de tal forma que se $a = b$, então f_a e g_b estão no mesmo grupo. Note que pode ser que tenhamos um conjunto de símbolos no mesmo grupo mesmo que não sejam relacionados por $=$. Por exemplo, se $a = b$ e $c = b$, então f_a e f_c podem estar no mesmo grupo mesmo que $a \neq c$ (a menos que $a = c$). Se $a \lessdot b$, então f_a e g_b não podem estar no mesmo grupo.
3. Crie um grafo dirigido cujos nós são os grupos criados em (2). Para cada a e b tais que $a \lessdot b$, coloque um arco do grupo de g_b para o grupo de f_a. Se $a \gtrdot b$, coloque um arco do grupo de f_a para o grupo de g_b. Note que um caminho do grupo de f_a para o grupo de g_b indica que f_a deve ser maior que g_b, e um caminho de g_b para f_a indica que g_b deve ser maior que f_a.
4. Se o grafo construído em (3) possuir um ciclo, então não existirão funções de precedência. Se não existir ciclo, faça $f(a)$ igual ao comprimento do mais longo caminho começando no grupo de f_a e faça $g(b)$ igual ao comprimento do mais longo caminho a partir do grupo de g_b.

Exemplo 4.30. Considere a matriz da Fig. 4.19. Nela existem quatro grupos: f_+ e g_+ formam um grupo, f_* e g_* formam outro, $f_($ e $g_)$ formam um terceiro, e f_{id}, g_{id}, $f_\$$ e $g_\$$ formam o quarto grupo. A Fig. 4.20 mostra o grafo construído pelo algoritmo 4.8.

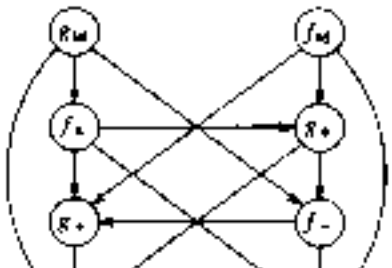


Fig. 4.20. Grafo representando funções de precedência.

Não existem ciclos, portanto as funções de precedência existem. Como $f_(\$$ não tem lados para fora, $f(()) = 0$. Os outros nós de maior grupo alcançável de $f_(\$$ possuem somente um lado para fora, portanto $f(+) = 1$, pois um lado leva ao grupo de g_+ para f_*, para g_* e para f_+, e, por conseguinte, $g(\text{id}) = 5$. As funções de precedência restantes são:

	+	*	id	$
f	2	4	4	0
g	1	3	5	0

Tratando as Operações Unárias

Se tivermos um operador unário, tal como $-$ (negação lógica), que também seja um operador binário, ou se tivermos operadores à direita, a relação de precedência pode ser determinada de acordo com as regras de precedência de operadores. Suponha que op_1 seja um operador unário prefixo. Fazemos $\text{op}_1 \lessdot a$ se op_1 tiver menor precedência do que a, e $\text{op}_1 \gtrdot a$ em caso contrário. Por exemplo, se op_1 tiver maior precedência do que $\&$ e $\&$ for um operador binário à esquerda, teremos $\text{op}_1 \gtrdot \&$ e $\& \lessdot \text{op}_1$ para todos os operadores $\&$. Vírgula para os operadores unários prefixos é tratada como vírgula final — operadores à esquerda aparecendo depois da vírgula têm precedência mais alta que a vírgula, e operadores à direita aparecendo antes da vírgula têm precedência mais baixa que a vírgula. Veja o Exercício 4.4.3(a) para uma alternativa de tratamento de unários.

Em muitos casos, é possível codificar uma matriz de precedência de operadores usando funções de precedência. Isto é, dados os símbolos a e b, podemos ter $f(a) < g(b)$ sempre que $a \lessdot b$, $f(a) = g(b)$ sempre que $a = b$, e $f(a) > g(b)$ sempre que $a \gtrdot b$. Pode-se constatar que a relação de precedência entre a e b pode ser determinada através de uma comparação numérica entre $f(a)$ e $g(b)$. Note, entretanto, que as entradas em branco na matriz de precedência são obstáculos, uma vez que impedem relações, e que funções de precedência sempre forçam uma relação para cada par. A perda da capacidade de detecção de erros nem sempre é considerada séria o suficiente para impedir o uso das funções de precedência, especialmente porque os erros também podem ser capturados quando uma redução for pedida e verificarmos que o lado direito não tem a forma correta.

Nem toda tabela de relações de precedência possui funções de precedência, mas existem muitos casos práticos. A função a seguir é útil:

Exemplo 4.29. A tabela de precedência da Fig. 4.19 possui os seguintes pares de funções de precedência:

	+	−	*	/	↑	(	)	id	$
f	2	2	4	4	4	0	6	6	0
g	1	1	3	3	5	5	0	5	0

Por exemplo, $2 < 3$ e $\text{id} \lessdot *$, e $4 > 3$ e $* \gtrdot \text{id}$. Note que $f(\text{id}) > g(\text{id})$ sugere que $\text{id} \gtrdot \text{id}$, mas de fato, nenhuma relação de precedência é suposta entre id e id. Outros exemplos de erro na Fig. 4.21 são similarmente substituídos por uma ou outra relação de precedência. ∎

Um método simples para se encontrar as funções de precedência para uma tabela de relações existentes é o seguinte.

Algoritmo 4.8. Construção de funções de precedência.

Entrada. Uma matriz de precedência de operadores.

Saída. Funções de precedência, ou uma indicação de que elas não existem.

Método

1. Crie os símbolos f_a e g_a para cada a que seja um terminal ou $\$$.
2. Particione os símbolos criados em tantos grupos quantos sejam possíveis de tal forma que se $a = b$, então f_a e g_b estão no mesmo grupo. Note que pode ser que tenhamos um conjunto de símbolos no mesmo grupo mesmo que não sejam relacionados por $=$. Por exemplo, se $a = b$ e $c = b$, então f_a e f_c podem estar no mesmo grupo mesmo que $a \neq c$ (a menos que $a = c$). Se $a \lessdot b$, então f_a e g_b não podem estar no mesmo grupo.
3. Crie um grafo dirigido cujos nós são os grupos criados em (2). Para cada a e b tais que $a \lessdot b$, coloque um arco do grupo de g_b para o grupo de f_a. Se $a \gtrdot b$, coloque um arco do grupo de f_a para o grupo de g_b. Note que um caminho do grupo de f_a para o grupo de g_b indica que f_a deve ser maior que g_b, e um caminho de g_b para f_a indica que g_b deve ser maior que f_a.
4. Se o grafo construído em (3) possuir um ciclo, então não existirão funções de precedência. Se não existir ciclo, faça $f(a)$ igual ao comprimento do mais longo caminho começando no grupo de f_a e faça $g(b)$ igual ao comprimento do mais longo caminho a partir do grupo de g_b.

Exemplo 4.30. Considere a matriz da Fig. 4.19. Nela existem quatro grupos: f_+ e g_+ formam um grupo, f_* e g_* formam outro, $f_($ e $g_)$ formam um terceiro, e f_{id}, g_{id}, $f_\$$ e $g_\$$ formam o quarto grupo. A Fig. 4.20 mostra o grafo construído pelo algoritmo 4.8.

Fig. 4.20. Grafo representando funções de precedência.

Não existem ciclos, portanto as funções de precedência existem. Como $f_(\$$ não tem lados para fora, $f(()) = 0$. Os outros nós de maior grupo alcançável de $f_(\$$ possuem somente um lado para fora, portanto $f(+) = 1$, pois um lado leva ao grupo de g_+ para f_*, para g_* e para f_+, e, por conseguinte, $g(\text{id}) = 5$. As funções de precedência restantes são:

	+	*	id	$
f	2	4	4	0
g	1	3	5	0

Recuperação de Erros na Análise Sintática de Precedência de Operadores

Existem dois pontos no processo de uma análise sintática de precedência de operadores onde um analisador sintático de precedência de operadores pode descobrir uma sintaxe

... se também a relação de igualdade entre o terminal mais à esquerda da pilha e a entrada for tal

2. Se uma *ação* for especificada, mas não se dá a prática se não for tal *ação* também for verdadeira

Relativamente ao algoritmo de análise sintática de precedência de operadores algorithm 4.3, parece reduzir *handles* constituídos somente por terminais. Entretanto, enquanto os não-terminais são também tratados momentaneamente, ainda precisamos ser capazes de localizar o mais à esquerda terminal. Então, ao falarmos em "o topo da pilha" queremos dizer o lado direito de uma produção, significa que os terminais serão os mesmos também as posições ocupadas por não-terminais.

Devemos observar que algoritmo 4.3 não faz nada para evitar pontos inseguros. Para tanto, precisamos de um conjunto de pilhas válidas para uma gramática em particular, exatamente como os passos 1) e 2) da Fig. 4.24. No algoritmo de análise sintática de precedência de operadores, estamos certos de encontrar uma relação $<\cdot$, $\doteq$ ou $\cdot>$ entre o símbolo mais alto da pilha e o próximo símbolo de entrada, mas essa relação não nos diz nada sobre o que pensar a respeito de uma pilha ou de uma sentença. Selecionamos, portanto, pares de símbolos adjacentes na pilha — ou seja, aqueles que estejam relacionados por $<\cdot$ ou $\doteq$. Por conseguinte, os passos 1) e 2) têm por base os símbolos adjacentes na pilha.

O fato de nos concentrarmos exatamente uma sequência de símbolos $X_{i+1} \ldots X_n$, $\ldots$ na pilha não significa que $X_1 \ldots X_i$ seja uma cadeia viável. Podemos ter descoberto algum problema durante a redução à direita. Consequentemente, temos uma quantidade de objetos que os passos 1) e 2) da Fig. 4.24 não identificam nos passos 1) e 2) de forma a determinar se a pilha contém ou não uma redução.

Tarefa: Testes Durante as Reduções

Podemos dividir a rotina de detecção e reconhecimento de erros em diversas partes. Uma delas ocorre durante a redução à direita. Por exemplo, essa rotina poderá remover da pilha os símbolos, exatamente como os passos 1) e 2) da Fig. 4.24. Entretanto, em vez de uma redução à direita ser bem-sucedida, ela poderá descobrir que, embora os símbolos adequados estejam na pilha, eles não estão relacionados da maneira correta. Para determinar o que está errado, devemos examinar a pilha que estiver sendo descompilada e também o símbolo de entrada. Por exemplo, suponhamos que não esteja sendo descompilada uma cadeia válida, porque não existe nenhuma produção que corresponda aos terminais do lado direito da pilha. Preferimos, então, consultar uma rotina de diagnóstico especial para descobrir o que há de errado. Por exemplo, se o lado direito de uma produção for $E \to E + E$, podemos emitir o diagnóstico

```
... ausente entre dois operandos
```

Poderemos também concluir que existe um lado direito bem-formado, mas sequência própria de terminais e não-terminais para o qual não existe redução. Por exemplo, se não houver descompilação em não-terminais intervenientes em envolvendo dois operandos, mas não houver um lado direito com dois operandos, poderíamos emitir o diagnóstico

```
... operador ausente
```

Aqui, *E* está no lugar de um não-terminal arbitrário, representado pelo não-terminal *E*. Por exemplo, se *if* e *then* fossem operadores, poderíamos emitir a mensagem "expressão ausente entre *if* e *then*".

Em geral, a dificuldade em se identificar os diagnósticos apropriados, que não pertencem ao lado esquerdo de uma produção, depende da existência de um número suficiente de pessoas cadeias que poderiam ser encontradas antes de $\$$. Suponha que β seja um lado direito de uma produção, mas β não tenha sido reduzido porque faltam alguns símbolos à direita. Suponha também que β seja o lado direito de mais de uma produção. Podemos, então, esperar que, se β for o lado direito correto, a redução será feita após a inserção de algum símbolo. Podemos, portanto, emitir uma mensagem sugerindo a inserção de qualquer símbolo que possa aparecer depois de β em uma produção. Se houver muitos símbolos desse tipo, podemos preferir uma mensagem mais geral.

Em geral, podemos descobrir muitos erros que poderiam ser descompilados durante o passo 1) da Fig. 4.24. São evidentes os erros gerados por cadeias que não podem ser reduzidas porque não existe nenhum lado direito para o qual se encontrem as cadeias. Outros procedimentos são possíveis quando não existem cadeias que possam ser descompiladas. Percorremos a pilha à procura de um símbolo X tal que X e o símbolo de entrada a estejam relacionados por $<\cdot$ ou $\doteq$. Em seguida, descompilamos a pilha até que o símbolo do topo tenha uma relação $<\cdot$ com o último símbolo descompilado. Se o erro for detectado durante a redução à direita, também poderemos descompilar um ou mais não-terminais, além de terminais, para encontrar um lado direito para o qual se possa reduzir.

Exemplo 4.31. Vamos supor que a gramática da Fig. 4.25 seja usada para a tradução de expressões infixas em expressões pós-fixas. Uma versão do procedimento `parse` para essa gramática é mostrada na Fig. 4.27. Existe somente um lado direito para cada não-terminal, e cada lado direito consiste em um único operando, um operador binário entre dois operandos ou um par de parênteses envolvendo uma expressão. Para simplificar, não mostramos o código para reconhecer operandos ou operadores. Em vez disso, `id` representa um operando, e `lexval` é o valor associado a esse operando. Os operadores são `+` e `*`.

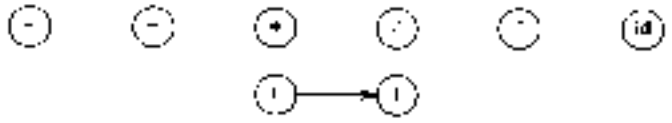


Fig. 4.27. Grafo de sintaxe abstrata para a gramática da Fig. 4.25.

Uma versão do procedimento `parse` para essa gramática é mostrada na Fig. 4.27. O código é gerado a partir do grafo de sintaxe abstrata mostrado na Fig. 4.27. Existe somente um lado direito para cada não-terminal, e cada lado direito consiste em um único operando, um operador binário entre dois operandos ou um par de parênteses envolvendo uma expressão. Para simplificar, não mostramos o código para reconhecer operandos ou operadores. Em vez disso, `id` representa um operando, e `lexval` é o valor associado a esse operando. Os operadores são `+` e `*`.

Uma versão do procedimento `parse` para essa gramática é mostrada na Fig. 4.27. O código é gerado a partir do grafo de sintaxe abstrata mostrado na Fig. 4.27. Existe somente um lado direito para cada não-terminal, e cada lado direito consiste em um único operando, um operador binário entre dois operandos ou um par de parênteses envolvendo uma expressão. Para simplificar, não mostramos o código para reconhecer operandos ou operadores. Em vez disso, `id` representa um operando, e `lexval` é o valor associado a esse operando. Os operadores são `+` e `*`.

Uma versão do procedimento `parse` para essa gramática é mostrada na Fig. 4.27. O código é gerado a partir do grafo de sintaxe abstrata mostrado na Fig. 4.27. Existe somente um lado direito para cada não-terminal, e cada lado direito consiste em um único operando, um operador binário entre dois operandos ou um par de parênteses envolvendo uma expressão. Para simplificar, não mostramos o código para reconhecer operandos ou operadores. Em vez disso, `id` representa um operando, e `lexval` é o valor associado a esse operando. Os operadores são `+` e `*`.

Igualmente, precisa-se garantir que nenhuma sub-sequência legítima de tokens de qualquer cadeia de entrada possa ser desempilhada. Se existir alguma, corre-se o mesmo risco de gerar uma mensagem de erro incorreta.

Se existir uma infinidade de cadeias que possam ser desempilhadas, as mensagens de erro não podem ser tabeladas — uma base, caso a caso. Poderíamos usar uma, como regra geral, para determinar se algum lado direito de produção está por ser desempilhado. Veja 2, mede a distância, medida em termos dos *tokens* inseridos, removidos ou modificados, entre a cadeia desempilhada e uma cadeia desempilhável. Em seguida, o diagnóstico especifica a suposição de que aquela produção era a pretendida... Se nenhuma produção estiver próxima à cadeia desempilhada, podemos emitir um diagnóstico geral, especificando que "alguma coisa está errada nesta linha".

Estratégia: Erros de Empilhar Redução

Precisamos agora descobrir a maneira pela qual a analisador sintático de precedência de operadores pode descobrir os erros. Ao consultar a matriz de precedência para decidir se empilha ou reduz (linhas 15 e 19 da Fig. 4.24), podemos encontrar que nenhuma relação existe entre o topo da pilha e o primeiro símbolo de entrada. Para exemplo, se pensamos que a e b sejam os dois símbolos de topo da pilha, b está no topo e a está logo abaixo dos dois símbolos de entrada e não existe relação de precedência entre b e c. Para recuperarmos, precisamos modificar a pilha, a entrada ou ambas. Podemos modificar símbolos inseridos na pilha ou remover símbolos de entrada ou da pilha. Se inserirmos ou modificarmos, precisamos ter cuidado para não criarmos uma laço infinito onde, por exemplo, inserimos permanentemente símbolos ao início da entrada sem estarmos capazes de reduzir ou empilhar quaisquer dos símbolos inseridos.

Uma técnica que não nos assegura a exclusão de laços infinitos é garantir que, após a recuperação, o símbolo corrente de entrada possa ser empilhado ou a entrada corrente torne-se $\$$, garantindo que nenhum símbolo seja colocado na entrada e a pilha seja eventualmente encurtada. Por exemplo, dado a no topo da pilha e c na entrada, se $a < c$, poderíamos desempilhar a. Outra estratégia é inserir na entrada ou no topo da pilha um símbolo tal que $a < b$ e $b < c$. Podemos fazer isso se encontrarmos um símbolo b tal que $a \doteq b$ e inserirmos b à frente de c na entrada. Mais geralmente, podemos inserir uma cadeia de símbolos tal que

$$a \lessdot b_1 \doteq b_2 \doteq \cdots \doteq b_n \gtrdot c$$

se a não for um símbolo para o qual não pode ser encontrado. A ação exata escolhida deverá refletir a noção do programador de como ele ou ela mais provavelmente cometeu o erro sintático.

Para cada entrada em branco na matriz de precedência, precisamos especificar uma rotina de recuperação de erros. A rotina pode ser chamada quando a configuração é $a < b$ ou $a \doteq b$ ou $a \gtrdot b$. Quando o analisador sintático encontra a entrada para a e b no passo 6(c) da Fig. 4.24 e não há relação de precedência alguma entre a e b, os mesmos métodos usados para a rotina de recuperação daquele erro

Outras duas dessas rotinas de tratamento de erro podem ser seguintes

e1: "Faltando operando"
    Insere id na entrada
    Emite diagnóstico "Faltando operando"
e2: "Faltando operando quando uma expressão começa por um parêntese à direita"
    Remove) da entrada
    Emite diagnóstico "Expressão vazia"
e3: "Faltando operador quando id ou) é seguido por id ou ("
    Insere + na entrada
    Emite diagnóstico "Faltando operador"
e4: "Faltando operando quando a expressão termina por um parêntese à esquerda"
    Desempilha (
    Emite diagnóstico "Expressão vazia"

Vamos considerar como esse mecanismo de tratamento de erro manipulará a entrada incorreta id +). As primeiras ações tomadas pelo analisador são empilhar id, reduzi-lo para E e empilhar $+$ e E para o não-terminal à direita. Na próxima ação, empilhar) e, então, a configuração

PILHA	ENTRADA
$\$E+$	) $\$$

Uma vez que $+ <$), uma redução é pedida e a função de recuperação de erros para reduções é requerida para inspecionar os E's à esquerda e à direita. Ao encontrar que algum desses E's está faltando, emite o diagnóstico

Faltando operando

e não faz qualquer alteração na configuração.
Nossa configuração é agora

PILHA	ENTRADA
$\$E+)$	$\$$

Não existe relação de precedência entre $+$ e), e a função e2 (Fig. 4.25) para este par de símbolos é e2. A rotina e2 provoca a impressão de diagnóstico

Expressão vazia

e remove o parêntese da entrada. Somos deixados com a configuração final abaixo. Empilhar $\$E+E$ e

PILHA	ENTRADA
$\$E+E$	$\$$

reduzir $E+E$ para E e, finalmente, empilhar $\$E$ e aceitar a entrada.

4.7 ANALISADORES SINTÁTICOS LR

Esta seção apresenta uma técnica eficiente de análise sintática ascendente que pode ser usada para decompor uma ampla classe de gramáticas livres de contexto. A técnica é chamada de análise sintática LR(k), ou LR, significando a varredura da entrada da esquerda para a direita (*left-to-right*), o "R" sendo uma derivação mais à direita (*rightmost derivation*) *de trás para a frente*, e o número de símbolos de entrada de *look-ahead* que são usados ao se tomar decisões na análise sintática. Quando (k) for omitido, é assumido ser 1. A técnica de análise sintática LR é atraente por uma série de razões:

- Analisadores sintáticos LR podem ser elaborados para reconhecer virtualmente todas as construções de linguagens de programação para as quais as gramáticas livres de contexto podem ser escritas.
- O método de decomposição LR é o mais geral dentre os métodos sem retrocesso ou empilhar e reduzir conhecidos ainda pode ser implementado tão eficientemente quanto os outros métodos de empilhar e reduzir.
- A classe de gramáticas que podem ser decompostas usando-se métodos LR é um superconjunto próprio da classe de gramáticas que podem ser decompostas usando-se analisadores sintáticos preditivos.
- Um analisador sintático LR pode detectar um erro sintático tão cedo quanto possível numa varredura da entrada da esquerda para a direita.

	id	(	)	$
id	e3	e3	$\gtrdot$	$\gtrdot$
(	$\lessdot$	$\lessdot$	$\doteq$	e4
)	e3	e3	$\gtrdot$	$\gtrdot$
$	$\lessdot$	$\lessdot$	e2	e1

Fig. 4.28. Matriz de precedência de operadores com entradas de erro.

A principal vantagem deste método está em ser muito trabalhoso construir manualmente para uma gramática típica de linguagem de programação os conjuntos necessários de itens LR. Felizmente, muitos dos geradores de analisadores LR disponíveis e descritos na próxima seção deste capítulo fazem exatamente isso. Dada uma gramática adequada, podemos às vezes usar um gerador de analisadores sintáticos para produzir automaticamente um analisador sintático para a mesma. Se a gramática estiver ambígua ou outras construções comuns de análise sintática de descendente recursiva forem encontradas, o gerador de analisadores pode localizar a fonte do problema para o projetista ou compilador responsável.

Após descrever a operação de um analisador LR, apresentamos três técnicas para construir uma tabela sintática para uma gramática. O primeiro método, chamado de LR simples (SLR — de *simple LR*), é o mais fácil de implementar, mas nem todas as gramáticas poderão ser testadas. Para ele funcionar, produzirá uma tabela sintática para algumas gramáticas em que as outras duas possam ter sucesso. O segundo método, chamado de LR canônico (ou LR), é o mais poderoso e o mais caro. O terceiro, chamado LR com antecipação (LALR — de *lookahead* LR), é de poder e custo intermediários. O método LALR na maioria das vezes para as gramáticas das linguagens de programação comuns, podendo ser implementado eficientemente. Algumas ferramentas geradoras de compiladores baseadas em LR são consideradas mais adiante neste capítulo.

O Algoritmo de Análise Sintática LR

A forma esquemática de um analisador sintático LR é mostrada na Fig. 4.20. Consiste em uma entrada, uma saída, uma pilha, um programa analisador e uma tabela sintática que possui duas partes (ação e desvio). O programa analisador é o mesmo para todos os tipos de analisadores LR; somente a tabela sintática muda de um analisador para outro. O programa de análise sintática lê os caracteres, um de cada vez, da entrada. Usa uma pilha para armazenar uma cadeia da forma $s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m$, onde s_m está no topo. Cada X_i é um símbolo gramatical e cada s_i um símbolo chamado de estado. Cada estado sumariza a informação contida na pilha abaixo dele e a combinação do símbolo de estado no topo da pilha e o símbolo corrente de entrada é usada para indexar a tabela sintática e determinar a decisão de empilhar ou reduzir que deve ser tomada. Numa implementação, os símbolos gramaticais não precisam ser mantidos na pilha e estarão incluídos aqui apenas para nos ajudar em nossas discussões. A parte de análise sintática do programa lê caracteres da entrada até que uma análise sintática seja encontrada ou até que ocorra um erro. Em cada passo, o programa determina s_m, o estado no topo da pilha, e a_i, o símbolo corrente de entrada. Em seguida, consulta a entrada $action[s_m, a_i]$ na tabela de ações sintáticas para o estado s_m e entrada a_i, que pode ter um dos quatro seguintes valores:

1. empilhar s, onde s é um estado;
2. reduzir através da produção gramatical $A \to \beta$;
3. aceitar; e
4. erro.

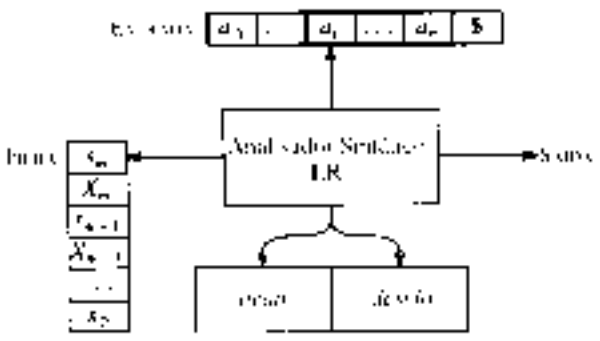


Fig. 4.20. Modelo de um analisador sintático LR.

A função *desvio* leva um estado e um símbolo gramatical como argumentos e produz um estado como saída. Veremos que a função *desvio* de uma tabela sintática construída a partir de uma gramática G é idêntica à função de transição de um autômato finito determinístico que reconhece os prefixos viáveis de G. Lembremos que os prefixos viáveis de G são aqueles das formas sentenciais à direita que podem aparecer na pilha de um analisador sintático de empilhar e reduzir porque não se estendem para além do *handle* mais à direita. O estado inicial deste AFD é o estado inicialmente colocado no topo da pilha do analisador sintático LR.

Uma *configuração* de um analisador sintático LR é um par cujo primeiro componente é o conteúdo da pilha e cujo segundo componente é a entrada ainda não consumida:

$$(s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m, a_i a_{i+1} \ldots a_n \$)$$

Esta configuração representa a forma sentencial à direita

$$X_1 X_2 \ldots X_m a_i a_{i+1} \ldots a_n$$

essencialmente da mesma maneira que faz um analisador com pilha e redução tática, somente a presença dos estados na pilha é mostrada aqui.

O próximo movimento do analisador é determinado pela leitura de a_i, o símbolo corrente da entrada, e s_m, o estado no topo da pilha, e pela consulta à entrada da tabela de ações sintáticas $action[s_m, a_i]$. As quatro configurações resultantes após cada um dos quatro tipos de movimento possíveis são como se segue:

1. Se $action[s_m, a_i] =$ empilhar s, o analisador executa um movimento de empilhar, entrando na configuração

$$(s_0 X_1 s_1 X_2 s_2 \ldots X_m s_m a_i s, a_{i+1} \ldots a_n \$)$$

Aqui o analisador sintático empilha tanto o símbolo corrente de entrada a_i quanto o próximo estado s que é dado por $action[s_m, a_i]$, se torna o símbolo corrente de entrada.

2. Se $action[s_m, a_i] =$ reduzir $A \to \beta$, então a análise sintática executa um movimento de redução, entrando na configuração

$$(s_0 X_1 s_1 X_2 s_2 \ldots X_{m-r} s_{m-r} A s, a_i a_{i+1} \ldots a_n \$)$$

onde $s = desvio[s_{m-r}, A]$ e r é o comprimento de β, o lado direito da produção. Aqui o analisador sintático remove primeiro $2r$ símbolos para fora da pilha (r símbolos de estado e r símbolos gramaticais), expondo o estado s_{m-r}. Em seguida empilha tanto A, o lado esquerdo da produção, quanto s, a entrada para $desvio[s_{m-r}, A]$. Para os analisadores sintáticos LR que iremos construir, $X_{m-r+1} \ldots X_m$, a sequência de r símbolos gramaticais no topo da pilha antes de sempre reconhecerá β, o lado direito da produção relevante.

A saída de um analisador sintático LR é gerada após um movimento de redução através da execução de uma ação semântica associada à produção reduzida. Para o momento, assumimos que a saída consiste somente na impressão da produção reduzida.

3. Se $action[s_m, a_i] =$ aceitar, a análise sintática está completa.
4. Se $action[s_m, a_i] =$ erro, o analisador sintático descobriu um erro e chama uma rotina de recuperação de erros.

O algoritmo de análise sintática LR é sumarizado abaixo. Todos os analisadores sintáticos LR se comportam desta maneira; a única diferença entre dois analisadores LR sintáticos é a informação nos campos de ação e desvio da tabela sintática.

Algoritmo 4.7. Algoritmo de análise sintática LR.

Entrada. Uma cadeia de entrada w e uma tabela sintática com funções *ação* e *desvio* para uma gramática G.

Saída. Se w estiver em $L(G)$, uma derivação *bottom-up* para w; caso contrário, uma indicação de erro.

Você pode iniciar um analisador sintático passando a pilha vazia e o estado inicial e lendo a função de entrada. O analisador sintático executa então o programa da Fig. 4.30 até que uma ação de aceitação ou de erro seja encontrada.

```
Deixe-o apontar para o primeiro símbolo de s;
repetir para sempre início
    seja s o estado no topo da pilha e
    a o símbolo apontado por ip;
    se ação[s, a] = empilhar s' então início
            empilhar a e em seguida s' no topo da pilha;
            avançar ip para o próximo símbolo de entrada
        fim
        senão se ação[s, a] = reduzir A → β então início
            desempilhar 2 * |β| símbolos da pilha;
            seja s' o estado agora no topo da pilha;
            empilhar A e em seguida ir[s', A];
            escrever a produção A → β
        fim
        senão se ação[s, a] = aceitar então
            retornar
        senão chamar-rotina-de-erro()
fim
```

Fig. 4.30. Programa de controle de um analisador LR.

Exemplo 4.73. A Fig. 4.31 mostra as funções de ação e desvio de uma tabela sintática LR para a seguinte gramática para expressões aritméticas com os operadores binários + e *:

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow \textbf{id}$

Deseja-se para cada função:

1. Se a função empilhar o símbolo de entrada mais o estado s;
2. Se a função reduzir através da produção de número y;
3. Se a função aceita;
4. Uma entrada em branco significa um erro.

Note que o valor de *desvio*[*s*, *a*] é só para um terminal *a*. É encontrado no campo *ação*[*s*, *a*], conectado à ação de empilhar a entrada *a* e estado *s*. O campo desvio é usado somente após uma redução. Uma redução de $A \rightarrow \beta$ faz com que |β| símbolos sejam desempilhados da pilha, revelando um estado *s'*. Igual nos analisadores sintáticos simples que analisamos, *desvio*[*s'*, *A*] é então adicionado à pilha.

Estado	ação					desvio		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Fig. 4.31. Tabela sintática para a gramática de expressões.

Pilha	Entrada	Ação
0	**id** * **id** + **id** $	empilhar
0 **id** 5	* **id** + **id** $	reduzir por $F \rightarrow$ **id**
0 F 3	* **id** + **id** $	reduzir por $T \rightarrow F$
0 T 2	* **id** + **id** $	empilhar
0 T 2 * 7	**id** + **id** $	empilhar
0 T 2 * 7 **id** 5	+ **id** $	reduzir por $F \rightarrow$ **id**
0 T 2 * 7 F 10	+ **id** $	reduzir por $T \rightarrow T * F$
0 T 2	+ **id** $	reduzir por $E \rightarrow T$
0 E 1	+ **id** $	empilhar
0 E 1 + 6	**id** $	empilhar
0 E 1 + 6 **id** 5	$	reduzir por $F \rightarrow$ **id**
0 E 1 + 6 F 3	$	reduzir por $T \rightarrow F$
0 E 1 + 6 T 9	$	reduzir por $E \rightarrow E + T$
0 E 1	$	aceitar

Fig. 4.32. Movimentos de um analisador sintático LR para **id** * **id** + **id**.

Para a entrada **id** * **id** + **id**, a sequência de conteúdos da pilha e da entrada é mostrada na Fig. 4.32. Por exemplo, a linha (1) mostra que o analisador LR está no estado 0, tendo **id** como o primeiro símbolo de entrada. A ação na linha (1) é empilhar o estado **id** do campo ação da Fig. 4.31 e o símbolo **id**. Após empilhar o símbolo de entrada **id** e estado 5, a linha (2) mostra a pilha com estado 5. Isso corresponde à configuração da linha (2) da Fig. 4.31. O próximo símbolo de entrada é * e o estado 5 na parte superior da pilha indica que **id** foi reconhecido como F pela função de ação.

Em segunda, consulta o símbolo corrente da entrada, *, e a ação do estado 5 para a entrada * é reduzir através de $F \rightarrow$ **id**. Os dois símbolos são removidos da pilha, um símbolo de estado e um símbolo gramatical. O estado 0 é descoberto. Uma vez que desvio[0, *F*] = 3, *F* e 3 são empilhados. Temos agora a configuração da linha (3). Cada uma das outras entradas restantes é determinada de forma similar.

Gramática LR

Como construímos uma tabela sintática LR para uma dada gramática? Uma gramática para a qual podemos construir uma tabela sintática LR é denominada uma *gramática LR*. Existem gramáticas livres de contexto que não são LR, mas que podem ser geralmente evitadas para as construções típicas das linguagens de programação. Intuitivamente, para uma gramática ser LR, é suficiente que um analisador sintático de empilhar e reduzir que, processando a entrada da esquerda para a direita, seja capaz de reconhecer os *handles* do lado direito, quando eles aparecem no topo da pilha.

Um analisador LR não precisa varrer toda a pilha para saber quando um *handle* surge no topo. De fato, o símbolo de estado no topo da pilha contém toda a informação necessária. Uma característica importante é que podemos reconhecer um *handle* conhecendo somente os símbolos gramaticais na pilha, e não os estados. Uma vez que podemos determinar qual *handle* se forma em algum estado, podemos associar ações a esse estado. Uma tabela sintática LR representa, essencialmente, uma máquina finita. O cuidado para preciso, entretanto, ler a pilha, cada movimento, o símbolo de estado associado ao topo da pilha. O estado que o autômato finito reconhece é o *handle* estar se tivesse todos os símbolos gramaticais na pilha. Isso nos permite usar estados em vez de símbolos gramaticais na pilha, pois, de fato, o estado armazena as informações necessárias obtidas da pilha inteira. O cuidado pelo preciso, entretanto, ler a pilha, cada movimento, o símbolo de estado associado ao topo da pilha. O estado que o autômato finito reconhece é o *handle* estar se tivesse todos os símbolos gramaticais na pilha. Isso nos permite usar estados em vez de símbolos gramaticais na pilha, pois, de fato, o estado armazena as informações necessárias obtidas da pilha inteira. O analisador sintático LR pode determinar, a partir do estado no topo da pilha, tudo o que precisa saber sobre o que está abaixo dele.

Uma outra fonte de informações que o analisador sintático LR pode usar para auxiliar na tomada de decisões de redução e empilhar são os próximos símbolos de entrada. Os casos $k = 0$ e $k = 1$ são de interesse prático, e nos restringiremos a eles. Se k não for especificado, assume-se que ele é igual a 1. Por exemplo, a tabela de ações da Fig. 4.31 não usa nenhuma informação de lookahead. Uma gramática que possa ser decomposta por um analisador sintático LR examinando até k símbolos de entrada a cada movimento é chamada de uma *gramática LR(k)*.

Existe uma significativa diferença entre uma gramática LL e uma LR. No caso de uma gramática ser LR(k), queremos ser capazes de reconhecer a ocorrência do lado direito de uma produção tendo visto tudo o que foi derivado a partir daquele lado direito mais *k* símbolos seguintes de entrada. Usualmente, *k* é 1 ou 2 símbolos de entrada. Essa exigência é muito menos restritiva do que aquela para uma gramática LL(*k*), onde precisamos ser capazes de reconhecer a ocorrência de uma produção examinando somente os primeiros *k* símbolos que serão derivados pelo lado direito. Dessa forma, as gramáticas LR podem descrever muito mais linguagens do que as gramáticas LL.

Construindo Tabelas Sintáticas SLR

Nesta seção, vamos agora mostrar como construir, a partir de uma gramática *G*, uma tabela sintática LR. Iremos fornecer três métodos que variam em sofisticação e na facilidade de sua implementação. O primeiro, chamado de "LR simples" ou SLR, abreviação de "LR simples", é o mais fácil de implementar, porém nem sempre o mais poderoso em termos do número de gramáticas para as quais ele pode ser usado com sucesso. Os outros dois métodos, chamados LR canônico e LALR, são mais trabalhosos de implementar, porém são aplicáveis a uma classe muito mais ampla de gramáticas. Todos os três métodos de construção de tabelas sintáticas LR diferem somente nos valores das entradas individuais na tabela; o algoritmo de análise sintática LR é o mesmo, independentemente do método escolhido. Em geral, nos referiremos à tabela sintática construída por esse método como uma tabela LR, ou tabela SLR, se o método for importante. Um analisador sintático LR que usa uma tabela sintática SLR é chamado analisador sintático SLR. Uma gramática para a qual um analisador sintático LR pode ser construído usando-se uma tabela SLR é chamada gramática SLR. Os métodos LR diferem somente na tabela sintática que é construída a partir de uma dada gramática, de forma que o método de análise sintática LR é o mesmo, não importando qual método é usado. Em todos os casos, ele lê a entrada da esquerda para a direita, um símbolo de cada vez. Em quase todos os casos, ele usa um método de redução de empilhamento de análise sintática ascendente, como discutido na Seção 4.5. Somente pequenas variações no algoritmo de análise sintática são necessárias para cada método. O que varia entre os métodos é o conteúdo específico das entradas da tabela sintática.

Antes de mostrar como construir uma tabela sintática SLR, precisamos introduzir algumas definições importantes e uma nova construção aplicada a uma gramática. Um *item LR(0)* (ou simplesmente item) de uma gramática *G* é uma produção de *G* com um ponto em algum lugar do lado direito. Por exemplo, a produção $A \to XYZ$ produz os quatro itens seguintes:

$$\begin{aligned} A &\to \cdot XYZ \\ A &\to X \cdot YZ \\ A &\to XY \cdot Z \\ A &\to XYZ \cdot \end{aligned}$$

A produção $A \to \varepsilon$ gera somente um item, $A \to \cdot$. Um item pode ser representado por um par de inteiros, o primeiro fornecendo o número de produção e o segundo a posição do ponto. Intuitivamente, um item indica quanto de uma produção nós encontramos a uma dada altura do processo de análise sintática. Por exemplo, o primeiro item acima indica que esperamos encontrar uma cadeia na entrada derivável a partir de XYZ. O segundo item indica que acabamos de examinar na entrada uma cadeia derivável a partir de X e que esperamos encontrar uma cadeia derivável a partir de YZ.

A idéia central no método SLR é construir primeiro, a partir da gramática em análise, um autômato finito determinístico que reconheça prefixos viáveis. Vamos agrupar esses itens em conjuntos, os quais dão origem aos estados do analisador sintático SLR. Os itens podem ser vistos como os estados de um AFD que reconhece prefixos viáveis. O agrupamento dos itens em conjuntos é o processo de construção de subconjuntos discutido na Seção 3.6.

Uma coleção de conjuntos de itens LR(0), que chamaremos uma coleção LR(0) canônica, provê a base para a construção de analisadores sintáticos SLR. Para construir a coleção canônica LR(0) para uma gramática, definimos uma gramática aumentada e duas funções, *fechamento* e *ir_para*.

Se *G* for uma gramática com símbolo de partida *S*, então *G'*, a gramática aumentada para *G*, é *G* com um novo símbolo de partida *S'* e uma produção $S' \to S$. O propósito desta nova produção de partida é o de indicar ao analisador sintático quando ele deve parar de analisar e anunciar a aceitação da entrada. Ou seja, a aceitação ocorre quando e somente quando o analisador sintático estiver para reduzir através de $S' \to S$.

Operação de Fechamento

Se *I* for um conjunto de itens para uma gramática *G*, então *fechamento(I)* é o conjunto de itens construídos a partir de *I* por essas duas regras:

1. Inicialmente, cada item em *I* é acrescentado ao *fechamento(I)*.
2. Se $A \to \alpha \cdot B\beta$ estiver em *fechamento(I)* e $B \to \gamma$ for uma produção, então acrescentar o item $B \to \cdot\gamma$ a *I*, se já não estiver lá. Aplicamos esta regra até que não possam ser adicionados mais itens ao *fechamento(I)*.

Intuitivamente, $A \to \alpha \cdot B\beta$ em *fechamento(I)* indica que, em algum ponto do processo de análise gramatical, esperamos poder ver em seguida na entrada uma subcadeia derivável a partir de $B\beta$. Se $B \to \gamma$ for uma produção, também esperamos ver uma subcadeia derivável de γ naquele ponto. Por esta razão também incluímos $B \to \cdot\gamma$ no *fechamento(I)*.

Exemplo 4.34. Consideremos a gramática de expressões aumentada

$$\begin{aligned} E' &\to E \\ E &\to E + T \mid T \\ T &\to T * F \mid F \\ F &\to (E) \mid \mathbf{id} \end{aligned} \qquad (4.19)$$

Se *I* for o conjunto de um item $\{[E' \to \cdot E]\}$, então o *fechamento(I)* contém os itens

$$\begin{aligned} E' &\to \cdot E \\ E &\to \cdot E + T \\ E &\to \cdot T \\ T &\to \cdot T * F \\ T &\to \cdot F \\ F &\to \cdot (E) \\ F &\to \cdot \mathbf{id} \end{aligned}$$

Aqui, $E' \to \cdot E$ é colocado em *fechamento(I)* pela regra 1. Como existe um *E* imediatamente à direita do ponto, pela regra 2 as produções-*E* com pontos nos seus lados direitos iniciais, $E \to \cdot E + T$ e $E \to \cdot T$, são acrescentadas. Em seguida, um *T* imediatamente à direita do ponto e consequentemente adicionamos $T \to \cdot T * F$ e $T \to \cdot F$. Em seguida, o *F* à direita do ponto força a inclusão de $F \to \cdot(E)$ e de $F \to \cdot \mathbf{id}$. Nenhum outro item é colocado no *fechamento(I)* pela regra 2. □

A função de *fechamento* pode ser computada como na Fig. 4.32. Uma forma conveniente de implementar a função é a de manter o booleano *adicionado*, indicando se algum item foi acrescentado ao conjunto *fechamento[I]* até o momento. Uma forma eficiente de computar *fechamento* é a de manter uma lista de itens ainda não adicionados a *fechamento[I]*, de tal forma que, quando um item $A \to \alpha \cdot B\beta$ for adicionado a *fechamento[I]*, somente os itens $B \to \cdot\gamma$, para cada produção $B \to \gamma$, precisam ser adicionados à lista.

```
função fechamento(I):
início
    J = I
    repetir
        para cada item A → α·Bβ em J e cada produção
            B → γ de G tal que B → ·γ não esteja em J faça
                incluir B → ·γ a J
    até que não possam ser adicionados mais itens a J.
    retornar J
fim
```

Fig. 4.32. Cômputo de *fechamento*.

Note que se uma produção-*B* for incluída no fechamento de *I*, com um ponto na extremidade esquerda, todas as produções-*B* serão similarmente incluídas. De fato, não é necessário, em certas circunstâncias, listar explicitamente os itens $B \to \cdot\gamma$ adicionados a *I* através de *fechamento*. Uma lista de não-terminais cujas produções foram adicionadas já basta. De fato, podemos dividir todos os conjuntos de itens que estejamos interessados em duas classes:

1. *Itens nucleares*, a qual inclui o item inicial $S' \to \cdot S$ e todos os demais itens cujos pontos não estejam na extremidade esquerda, e
2. *Itens não-nucleares*, a qual inclui todos os itens que tenham o ponto na extremidade esquerda.

Sobretudo, cada conjunto de itens em que estejamos interessados é formado tomando-se um conjunto de itens nucleares e os itens adicionados

...

O fecho(...)

...

Exemplo 4.35. Se I for o conjunto de itens $\{E' \to \cdot E\}$, então fecho(I) será o conjunto de itens:

$$\begin{aligned} E' &\to \cdot E \\ E &\to \cdot E + T \\ E &\to \cdot T \\ T &\to \cdot T * F \\ T &\to \cdot F \\ F &\to \cdot (E) \\ F &\to \cdot \mathbf{id} \end{aligned}$$

O conjunto de itens fecho(I) é formado do item original com o ponto imediatamente à direita do ponto, $E' \to \cdot E$, mais os itens $E \to \cdot E + T$ e $E \to \cdot T$. Movendo o ponto por sobre E e T para obtermos $E' \to E \cdot$ e $E \to E \cdot + T$, e em seguida realizando o fechamento deste conjunto, ...

A construção dos conjuntos de itens

Estamos agora prontos para ver como a gramática pode ser usada para construir a coleção canônica de conjuntos de itens LR(0) para uma gramática aumentada. O algoritmo a seguir é mostrado na Fig. 4.34.

```
procedimento itens(G'):
início
    C = { fecho({S' → ·S}) };
    repita
        para cada conjunto de itens I em C e cada símbolo
            gramatical X tal que I → X) está em C mas não está em C
                faça inclua I → X) em C
    até que não haja mais conjuntos de itens a serem incluídos em C
fim
```

Fig. 4.34. Algoritmo para construir a coleção canônica de conjuntos de itens.

Exemplo 4.36. A coleção canônica de conjuntos de itens LR(0) para a gramática (4.1) do Exemplo 4.34 é mostrada na Fig. 4.35. A função ir_a pode ser representada pelas transições mostradas entre os estados; ela é também mostrada na tabela de transições da Fig. 4.36. ...

Se colocarmos o estado I_0 da Fig. 4.36 em um estado final, I_1 em estado de aceitação e cada I_i em estado i, obteremos exatamente os prefixos viáveis da gramática (4.18). Isto não é coincidência. Para cada gramática G, a função ir_a sobre a coleção canônica de conjuntos de itens de G é um diagrama de transições que reconhece os prefixos viáveis de G. De fato, podemos usar qualquer conjunto de itens como estado inicial, desde que restrinjamos os estados acessíveis aos prefixos viáveis que reconhecem. Há uma transição de $A \to \alpha \cdot X \beta$ para $A \to \alpha X \cdot \beta$, rotulada X, e existe uma transição de $A \to \alpha \cdot B \beta$ para $B \to \cdot \gamma$, rotulada ε. Então, o fechamento ε do conjunto de itens resultado de X é exatamente o fecho_lr0 sobre o conjunto de itens que consiste em $A \to \alpha X \cdot \beta$. Por conseguinte, dois conjuntos de itens I_1 e I_2 podem ser unidos se I_1 for acessível a partir de I_2 mediante X e I_2 for acessível a partir de I_1 mediante X. Isto é exatamente o que acontece com os conjuntos I_3 e I_{10} da Fig. 4.36. Por exemplo, I_3 é $E \to E + \cdot T$, $T \to \cdot T * F$, $T \to \cdot F$, $F \to \cdot (E)$ e $F \to \cdot \mathbf{id}$, enquanto I_{10} é $E \to E + T \cdot$ e $T \to T \cdot * F$. Nenhum item em I_3 tem T como seu primeiro símbolo gramatical à direita do ponto, e nenhum item em I_{10} tem F como seu primeiro símbolo gramatical à direita do ponto. Assim, I_3 e I_{10} podem ser unidos. O resultado é o conjunto de itens $I_{3,10}$:

$$\begin{aligned} E &\to E + \cdot T \\ E &\to \cdot E + T \\ E &\to \cdot T \\ T &\to \cdot T * F \\ T &\to \cdot F \\ F &\to \cdot (E) \\ F &\to \cdot \mathbf{id} \end{aligned}$$

I_4:
$$\begin{aligned} F &\to (\cdot E) \\ E &\to \cdot E + T \\ E &\to \cdot T \\ T &\to \cdot T * F \\ T &\to \cdot F \\ F &\to \cdot (E) \\ F &\to \cdot \mathbf{id} \end{aligned}$$

I_5:
$$F \to \mathbf{id} \cdot$$

I_6:
$$\begin{aligned} E &\to E + \cdot T \\ T &\to \cdot T * F \\ T &\to \cdot F \\ F &\to \cdot (E) \\ F &\to \cdot \mathbf{id} \end{aligned}$$

I_7:
$$\begin{aligned} T &\to T * \cdot F \\ F &\to \cdot (E) \\ F &\to \cdot \mathbf{id} \end{aligned}$$

I_8:
$$\begin{aligned} F &\to (E \cdot) \\ E &\to E \cdot + T \end{aligned}$$

I_9:
$$\begin{aligned} E &\to E + T \cdot \\ T &\to T \cdot * F \end{aligned}$$

I_{10}:
$$T \to T * F \cdot$$

I_{11}:
$$F \to (E) \cdot$$

Fig. 4.35. Coleção canônica LR(0) para a gramática (4.19).

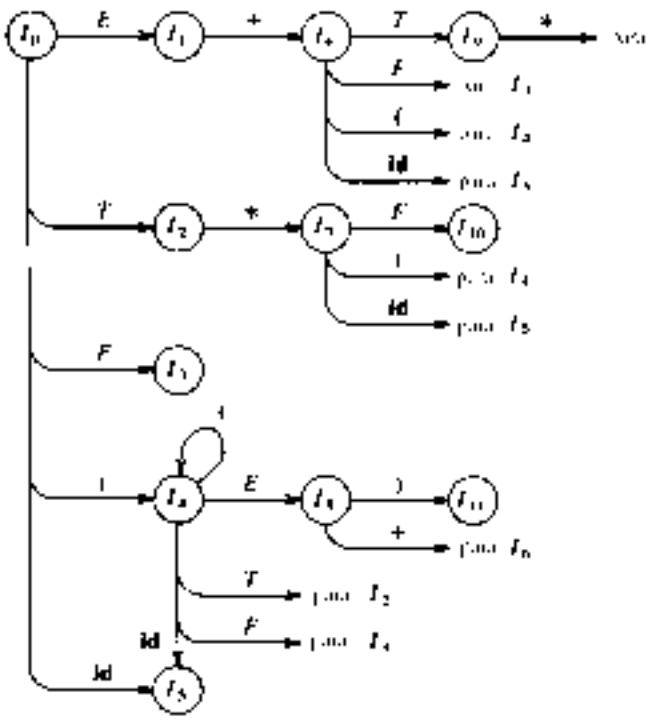


Fig. 4.36. Diagrama de transições para o AFD da Fig. 4.35 e tabela de transições para a função ir_a.

Prefixos viáveis. Dizemos que o item $A \to \beta_1 \cdot \beta_2$ é *válido* para o prefixo viável $\alpha\beta_1$ se existir uma derivação $S' \Rightarrow^* \alpha A w \Rightarrow \alpha\beta_1\beta_2 w$. Em geral, um mesmo item será válido para muitos prefixos viáveis. O item $A \to \beta_1 \cdot \beta_2$ será válido para $\alpha\beta_1$ independentemente de qual seja a cadeia α que o precede, ou seja, independentemente do estado que o conjunto $\alpha\beta_1$ ocupa no AFD da coleção canônica. Em particular, se $\beta_2 = \varepsilon$, então $A \to \beta_1 \cdot$ indica que $A \to \beta_1$ é um *handle*, e podemos reduzir através desta produção. Na...

Consequentemente, cada célula pode conter duas diferentes ações, sendo uma delas o conflito. Algumas dessas condições podem ser resolvidas examinando-se o próximo símbolo de entrada e os demais conflitos podem ser tratados de métodos desenvolvidos nesta seção. Mas não devemos supor que todos os conflitos das ações da análise sintática possam ser resolvidos se o método LR for usado para construir uma tabela sintática para uma gramática arbitrária.

Podemos facilmente computar o conjunto de itens válidos para cada prefixo viável que possa aparecer na pilha de um analisador sintático LR. De fato, é um teorema essencial da teoria da análise sintática LR que o conjunto de itens válidos para um prefixo viável γ seja exatamente o conjunto de itens alcançados a partir do estado inicial, pelo caminho com nome γ. Consequentemente, um AFD construído a partir do coleção canônica de conjuntos de itens, com as transições dadas pela função *ir_ir*, pode ser usado para reconhecer prefixos viáveis. Um conjunto de itens válidos incorpora todas as informações úteis que podem ser armazenadas na pilha. Conquanto não verifiquemos isso formalmente, aqui daremos um exemplo.

Exemplo 4.37. Vamos considerar a gramática (4.19) de novo, cujos conjuntos de itens válidos foram descritos exaustivamente nas Figs. 4.35 e 4.36. Um exemplo, a saber, $E + T * F$ é um prefixo viável de (4.19). Autômato finito da Fig. 4.36 estará no estado I_7 após ter lido $E + T *$. O estado I_7 contém os itens

$$\begin{aligned} T &\to T * \cdot F \\ F &\to \cdot (E) \\ F &\to \cdot \mathbf{id} \end{aligned}$$

que são precisamente os itens válidos para $E + T *$. Para ver isso, considere as três seguintes derivações à direita:

$$\begin{aligned} E &\Rightarrow E \\ &\Rightarrow E + T \\ &\Rightarrow E + T * F \end{aligned} \qquad \begin{aligned} E &\Rightarrow E \\ &\Rightarrow E + T \\ &\Rightarrow E + T * (E) \end{aligned} \qquad \begin{aligned} E &\Rightarrow E \\ &\Rightarrow E + T \\ &\Rightarrow E + T * \mathbf{id} \end{aligned}$$

A primeira derivação mostra a validade de $E + T * F$ a segunda a validade de $E + T * (E$ e a terceira a validade de $E + T * \mathbf{id}$ para o prefixo viável $E + T *$. Pode-se demonstrar que não existem outros itens válidos para $E + T *$, e deixaremos a prova para o leitor interessado. □

Tabelas Sintáticas SLR

Agora estamos prontos para construir as funções de ação e desvio para um analisador sintático que reconhece os prefixos viáveis. Nosso algoritmo não irá produzir tabelas de ações sintáticas unicamente definidas para toda e qualquer gramática, mas terá sucesso em muitas gramáticas e para linguagens de programação. Dada uma gramática G, iremos aumentá-la de forma a produzir G' e a partir de G' construiremos C, a coleção canônica de conjuntos de itens para G'. Construiremos a função de ações sintáticas *ação* a partir de C. Os passos do algoritmo a seguir são explicados a seguir. Frequentemente chamaremos SEGUINTE(*A*) para cada não terminal A da gramática (ver a Seção 4.4).

Algoritmo 4.46. *Construção de uma tabela sintática SLR.*

Entrada. Uma gramática aumentada G'.

Saída. As funções sintáticas SLR *ação* e *desvio* para G'.

Método.

1. Construa $C = \{I_0, I_1, \ldots, I_n\}$, a coleção de conjuntos de itens LR(0) para G'.
2. O estado i é construído a partir de I_i. As ações sintáticas para o estado i são determinadas como se segue:
   a) Se $[A \to \alpha \cdot a\beta]$ estiver em I_i e $ir\_ir(I_i, a) = I_j$, então estabeleça *ação*$[i, a]$ como "empilhar j". Aqui a precisa ser um terminal.
   b) Se $[A \to \alpha \cdot]$ estiver em I_i, então estabeleça *ação*$[i, a]$ como "reduzir $A \to \alpha$" para todo a em SEGUINTE(A); aqui A não pode ser S'.
   c) Se $[S' \to S \cdot]$ estiver em I_i, então estabeleça *ação*$[i, \$]$ como "aceitar".

Se quaisquer ações conflitantes forem geradas pelas regras anteriores, diz-se que a gramática não é SLR(1). O algoritmo falha em produzir um analisador sintático neste caso.

3. As transições de desvio para o estado i são construídas, para todos os não terminais A, usando-se a seguinte regra: se *desvio*$(I_i, A) = I_j$, então *desvio*$[i, A] = j$.
4. Todas as entradas não definidas pelas regras (2) e (3) são tornadas "erro".
5. O estado inicial do analisador sintático é aquele construído a partir do conjunto de itens contendo $[S' \to \cdot S]$.

A tabela sintática consistindo das funções sintáticas de ação e desvio determinada pelo algoritmo (4.46) é chamada de tabela SLR(1) para G. Um analisador sintático LR usando uma tabela SLR(1) para G é denominado analisador sintático SLR(1) para G, e uma gramática tendo uma tabela sintática SLR(1) é dita ser SLR(1). Usaremos frequentemente a abreviação SLR(1) para "SLR(1)" e "SLR" para "SLR(1)". Quando omitirmos o "1" após o símbolo LR, uma vez que não trataremos conosco analisadores sintáticos que tenham mais de um símbolo de lookahead.

Exemplo 4.38. Vamos construir a tabela SLR para a gramática (4.19). A coleção canônica de conjuntos de itens LR(0) para (4.19) foi mostrada na Fig. 4.35. Consideremos primeiro o conjunto de itens I_0:

$$\begin{aligned} E' &\to \cdot E \\ E &\to \cdot E + T \\ E &\to \cdot T \\ T &\to \cdot T * F \\ T &\to \cdot F \\ F &\to \cdot (E) \\ F &\to \cdot \mathbf{id} \end{aligned}$$

Todos $E \to \cdot$ (da origem para o estado 4 gera *ação*$[0, (] =$ empilhar 4. O item $F \to \cdot \mathbf{id}$ coloca *ação*$[0, \mathbf{id}] =$ empilhar 5. Os outros itens em I_0 não produzem ações. Agora consideremos I_1:

$$\begin{aligned} E' &\to E \cdot \\ E &\to E \cdot + T \end{aligned}$$

O primeiro item produz *ação*$[1, \$] =$ aceitar, enquanto produz *ação*$[1, +] =$ empilhar 6. Agora consideremos I_2:

$$\begin{aligned} E &\to T \cdot \\ T &\to T \cdot * F \end{aligned}$$

Uma vez que SEGUINTE(E) = $\{\$, +,)\}$, o primeiro item impõe ação$[2, \$] =$ *ação*$[2, +] =$ *ação*$[2,)]$ reduzir $E \to T$. O segundo item produz *ação*$[2, *] =$ empilhar 7. Continuando desta forma, produzimos as tabelas sintáticas de ações e de desvio que foram mostradas na Fig. 4.31. Naquela figura, os números de produções nas ações de redução são os mesmos que os da ordem na qual aparecem na gramática (4.19) original. Isto é, $E \to E + T$ é o número 1, $E \to T$ é 2, e assim por diante. □

Exemplo 4.39. Toda gramática SLR(1) é inambígua, mas existem muitas gramáticas inambíguas que não são SLR(1). Consideremos a gramática com as produções

$$\begin{aligned} S &\to L = R \\ S &\to R \\ L &\to * R \\ L &\to \mathbf{id} \\ R &\to L \end{aligned} \tag{4.20}$$

Podemos pensar de L e R como variáveis representando "valor à esquerda" e "valor à direita", respectivamente, e * como um operador indicando "conteúdo de". A coleção canônica

---

1 Isto é, seja C a coleção de conjuntos de itens válidos. Então, antes de novas entradas serem consideradas, a pilha terá γ em algum momento se, e somente se, γ for o rótulo de algum caminho do estado inicial para algum estado em C.

de conjuntos de itens LR(0) para a gramática (4.20) é mostrada na Fig. 4.37.

I_0: $S' \to \cdot S$
$S \to \cdot L = R$
$S \to \cdot R$
$L \to \cdot *R$
$L \to \cdot \mathbf{id}$
$R \to \cdot L$

I_1: $S' \to S \cdot$

I_2: $S \to L \cdot = R$
$R \to L \cdot$

I_3: $S \to R \cdot$

I_4: $L \to * \cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot \mathbf{id}$

I_5: $L \to \mathbf{id} \cdot$

I_6: $S \to L = \cdot R$
$R \to \cdot L$
$L \to \cdot *R$
$L \to \cdot \mathbf{id}$

I_7: $L \to *R \cdot$

I_8: $R \to L \cdot$

I_9: $S \to L = R \cdot$

Fig. 4.37. Coleção canônica LR(0) para a gramática (4.20).

Consideremos o conjunto de itens I_2. O primeiro item neste conjunto faz com que ação[2, =] seja "empilhar 6". Uma vez que SEGUINTE(R) contém =, para ver por que, considere $S \Rightarrow L = R \Rightarrow *R = R$, o segundo item $L \to R \cdot$ faz com que ação[2, =] seja também "reduzir por $L \to R$". Dessa forma, a entrada ação[2, =] é multiplamente definida. Claramente, existe tanto uma entrada de empilhar quanto uma de reduzir em ação[2, =], o estado 2 possui um conflito de empilhar/reduzir no símbolo de entrada =.

A gramática (4.20) não é ambígua. O conflito de empilhar/reduzir em ação de fato diz o método de construção do analisador sintático SLR não será poderoso o suficiente para se lembrar o contexto à esquerda de forma a decidir qual ação o analisador sintático deverá tomar à entrada =, tendo examinado tão somente a variável L. Os métodos canônico LALR discutidos em seguida funcionam sobre uma coleção mais ampla de gramáticas, incluindo a gramática (4.20). Deveria ser assinalados, entretanto, que existem gramáticas ambíguas para as quais qualquer método de construção de analisadores sintáticos LR irá produzir tabelas sintáticas com conflitos nas ações sintáticas. Infelizmente, tais gramáticas não são geralmente evitadas nas aplicações de linguagens de programação.

Construindo Tabelas Sintáticas LR Canônicas

Vamos agora apresentar o método mais geral para construir uma tabela sintática LR a partir de uma gramática. Relembremos que, no método SLR, o estado i chama pela redução por $A \to \alpha$ se o conjunto de itens I_i contiver o item $[A \to \alpha \cdot]$ e a entrada a estiver em SEGUINTE(A). Em algumas situações, entretanto, quando o estado i aparece no topo da pilha, o prefixo viável $\beta\alpha$ na pilha será tal que βA não poderia ser seguido por a numa forma sentencial à direita. Por conseguinte, uma redução para $A \to \alpha$ seria inválida à entrada a.

Exemplo 4.40. Vamos reconsiderar o Exemplo 4.39, onde, no estado 2, tínhamos o item $R \to L \cdot$, que corresponderia a $A \to \alpha \cdot$ acima, e a poderia ser o sinal de = que aparece em SEGUINTE(R). Dessa forma, o analisador sintático SLR chama pela redução por $R \to L$ no estado 2 com = como a próxima entrada (ação de empilhar também é chamada por causa do item $S \to L = R$ no estado 2). Entretanto, não existe forma sentencial à direita na gramática do Exemplo 4.39 que comece com $R =$. Consequentemente, o estado 2, que é o estado correspondente ao prefixo viável L somente, não deveria realmente chamar pela redução daquele L para R.

É possível se carregar tais informações mais exatas, se quisermos, mas existem algumas dessas reduções chamadas por $A \to \alpha$ para as quais os estados, quando necessário, podemos arranjar para ter cada estado de um analisador sintático LR indicando exatamente quais símbolos de entrada podem seguir a uma A válido para a qual exista uma possível redução para A.

A informação extra é incorporada dentro do estado pela redefinição dos itens, de forma a que incluam um símbolo terminal como segundo componente. A forma geral de um item se torna $[A \to \alpha \cdot \beta, a]$, onde $A \to \alpha\beta$ é uma produção e a é um terminal ou o marcador de fim à direita. Chamamos a um tal objeto um *item LR(1)*. O "1" se refere ao comprimento do segundo componente, chamado de *conjunto de lookahead*.[1] Esses *lookaheads* não possuem efeito numa item da forma $[A \to \alpha \cdot \beta, a]$, onde β não seja ε, mas um item da forma $[A \to \alpha \cdot, a]$ chama pela redução por $A \to \alpha$ somente se o próximo símbolo de entrada for a. Por conseguinte, temos um conjunto de itens válidos para cada estado, e todos os itens $[A \to \alpha \cdot, a]$ em um item LR(1) no estado ao topo da pilha. O conjunto de itens será um poderoso suficiente para se saber se um R que aparece em uma forma sentencial à direita deve ser reduzido para L, de forma que o conflito de ação no exemplo anterior desapareça.

Formalmente, dizemos que um item LR(1) $[A \to \alpha \cdot \beta, a]$ é *válido* para um prefixo viável γ se existir uma derivação à direita $S \Rightarrow_{rm}^{*} \delta Aw \Rightarrow_{rm} \delta\alpha\beta w$, onde

1. $\gamma = \delta\alpha$, e
2. a é o primeiro símbolo de w ou $w = \varepsilon$ e $a = \$$.

Exemplo 4.41. Vamos considerar a gramática

$$S \to BB$$
$$B \to aB \mid b$$

Existe uma derivação à direita $S \Rightarrow_{rm} aaBab \Rightarrow_{rm} aaaBab$. Fazendo $\delta = aa$, $A = B$, $w = ab$, $\alpha = a$ e $\beta = B$ na definição acima, vemos que existe $[B \to a \cdot B, a]$ válido para o prefixo viável $\gamma = aaa$. Existe também uma derivação à direita $S \Rightarrow_{rm} BaB \Rightarrow_{rm} BaaB$. A partir desta derivação podemos ver que o item $[B \to a \cdot B, \$]$ é válido para o prefixo viável Baa. □

Há várias possíveis construções da coleção de conjuntos de itens LR(1) válidos, e essencialmente somente aquela com que começaremos é conhecida como a coleção canônica de conjuntos de itens LR(1). Neste capítulo, apenas modificaremos os métodos precedentes para construir o DFA.

Para apresentarmos a nova definição da operação de *fechamento*, consideremos um item da forma $[A \to \alpha \cdot B\beta, a]$ no conjunto de itens válidos para algum prefixo viável γ. Existe, então, uma derivação à direita $S \Rightarrow_{rm}^{*} \delta Aax \Rightarrow_{rm} \delta\alpha B\beta ax$, onde $\gamma = \delta\alpha$. Suponhamos que βax derive-se em by, onde b é um terminal. Então, para cada produção da forma $B \to \eta$ para algum η, temos a derivação $S \Rightarrow_{rm}^{*} \gamma Bby \Rightarrow_{rm} \gamma\eta by$. Consequentemente, $[B \to \cdot\eta, b]$ é válido para γ. Notemos que b pode ser o primeiro terminal derivado a partir de β, ou é possível que β derive-se em ε na derivação $\beta ax \Rightarrow_{rm}^{*} by$, e b pode, consequentemente, ser a. Para sumarizar ambas as possibilidades, dizemos que b pode ser qualquer terminal em PRIMEIRO(βa), onde PRIMEIRO é a função da Seção 4.4. Notemos que a não pode conter o primeiro terminal de by, e então PRIMEIRO(βa) = PRIMEIRO(β). Fornecemos agora a construção de conjuntos de itens LR(1):

Algoritmo 4.9. Construção dos conjuntos de itens LR(1).

Entrada. Uma gramática aumentada G'.

Saída. Os conjuntos de itens LR(1) que são os conjuntos de itens válidos para um ou mais prefixos viáveis de G'.

Método. O principal instrumento é a função *fecho*, cuja rotina principal, para construir os conjuntos de itens, são mostrados na Fig. 4.38.

[1] Todos os conjuntos de lookahead são conjuntos não vazios de terminais e provavelmente serão finitos em todas as situações práticas.

```
função ir_para(I, X)
início
    repetir
        para cada item [A → α•Bβ, a] em I
            e cada produção B → γ em G',
            e cada terminal b em PRIMEIRO(βa)
            tal que [B → •γ, b] não está em I faça
            incluir [B → •γ, b] em I;
    até que não possam ser adicionados mais itens a I;
    retornar I
fim;

função itens(G')
início
        seja C o conjunto de itens {[A → αA•β, a]} tais que
            [A → α•Aβ, a] esteja em C;
        retornar o fecho(C)
fim;

procedimento ITENS(G')
início
    C := {fecho({[S' → •S, $]})};
    repetir
        para cada conjunto de itens I em C e cada símbolo
            gramatical X tal que ir_para(I, X) não esteja
            em C e ir_para(I, X) não seja vazio faça
            incluir ir_para(I, X) em C;
    até que não possam ser adicionados itens a C
fim
```

Fig. 4.38 Construção de conjuntos de itens LR(1) para a gramática G'

Exemplo 4.42. Considere-se a seguinte gramática aumentada.

$$\begin{array}{lll} S' \to S & & (1) \\ S \to CC & & (2) \\ C \to cC \mid d & & (3) \end{array}$$

Começamos pelo conjunto de itens fechamento de $\{[S' \to \bullet S, \$]\}$. Para fechar, avaliamos o item $[S' \to \bullet S, \$]$ com o item $[A \to \alpha\bullet B\beta, a]$ no procedimento fecho acima. Obtemos $A = S'$, $\alpha = \varepsilon$, $B = S$, $\beta = \varepsilon$ e $a = \$$. A função nos manda pôr em fecho(I) todos os itens $[B \to \bullet\gamma, b]$ para cada produção $B \to \gamma$ e terminal b em PRIMEIRO(βa). Uma vez que β e a são ε e $\$$, respectivamente, e há uma produção $S \to CC$, acrescentamos $[S \to \bullet CC, \$]$. Continuando a computar o fechamento, adicionamos todos os itens $[C \to \bullet cC, b]$ para b em PRIMEIRO($C\$$). Isto é, c e d, uma vez que $[C \to \bullet cC, c]$ e $[C \to \bullet cC, d]$ têm a forma $[B \to \bullet\gamma, b]$ com $B = C$, $\gamma = cC$ e $b = c$ ou d. Como C não deriva a cadeia vazia, PRIMEIRO($C\$$) = PRIMEIRO(C). Uma vez que PRIMEIRO(C) contém somente c e d, adicionamos os itens $[C \to \bullet cC, c]$, $[C \to \bullet cC, d]$, $[C \to \bullet d, c]$ e $[C \to \bullet d, d]$. Nenhum dos novos itens possui um não-terminal imediatamente à direita do ponto, e então completamos nosso primeiro conjunto de itens LR(1). O conjunto inicial de itens é

$$\begin{array}{ll} I_0: & S' \to \bullet S, \$ \\ & S \to \bullet CC, \$ \\ & C \to \bullet cC, c/d \\ & C \to \bullet d, c/d \end{array}$$

Os símbolos foram agrupados por conveniência de notação, e usamos o item $[C \to \bullet cC, c/d]$ como uma abreviação para os dois itens $[C \to \bullet cC, c]$ e $[C \to \bullet cC, d]$.

Computamos agora $ir\_para(I_0, X)$ para os vários valores de X. Para $X = S$ avaliamos fecho de $[S' \to S\bullet, \$]$. Nenhum fechamento adicional é possível, uma vez que o ponto está à extremidade direita. Por conseguinte, temos o próximo conjunto de itens

$$I_1: \quad S' \to S\bullet, \$$$

Para $X = C$, fecho de $[S \to C\bullet C, \$]$. Adicionamos as produções C com segundo componente $\$$ e então, não podemos adicionar mais, produzindo

$$\begin{array}{ll} I_2: & S \to C\bullet C, \$ \\ & C \to \bullet cC, \$ \\ & C \to \bullet d, \$ \end{array}$$

Em seguida, seja $X = c$. Precisamos fechar $[C \to c\bullet C, c/d]$. Adicionamos as produções C com o segundo componente c/d, produzindo

$$\begin{array}{ll} I_3: & C \to c\bullet C, c/d \\ & C \to \bullet cC, c/d \\ & C \to \bullet d, c/d \end{array}$$

Finalmente, para $X = d$ descobrimos o conjunto de itens

$$I_4: \quad C \to d\bullet, c/d$$

Terminamos de considerar *ir_para* com I_0. Não vemos conjuntos de I_0 mas I_1 possui *ir_para* em S e em $\$$, e assim por diante.

$$I_5: \quad S' \to CC\bullet, \$$$

e nenhum fechamento é necessário, pois o ponto vem ao fechamento de $[C \to c\bullet C, \$]$, produzindo

$$\begin{array}{ll} I_6: & C \to c\bullet C, \$ \\ & C \to \bullet cC, \$ \\ & C \to \bullet d, \$ \end{array}$$

Note-se que I_6 difere de I_3 somente nos segundos componentes. Veremos que é comum vários conjuntos de itens LR(1) de uma gramática terem os mesmos primeiros componentes e diferirem nos segundos. Quando estudarmos a coleção de conjuntos de itens LR(0) para nossa gramática, cada conjunto de itens LR(0) irá combinar com o conjunto de primeiros componentes de um ou mais conjuntos de itens LR(1). Veremos mais adiante que este fenômeno ocorre quando discutirmos análise sintática LALR.

Continuando com a função *ir_para*, $ir\_para(I_3, c) = I_3$, o que acabou ser

$$I_7: \quad C \to cC\bullet, \$$$

Voltando agora para I_4, os *ir_para* de I_4 para c e d são I_3 e I_4, respectivamente, e $ir\_para(I_4, C)$ é

$$I_8: \quad C \to cC\bullet, c/d$$

I_5 e I_8 não possuem *ir_para*. Os *ir_para* de I_6 em c e d são I_6 e I_7, respectivamente, e *ir_para* de I_8 em C é

$$I_9: \quad C \to cC\bullet, \$$$

Os conjuntos remanescentes de itens não produzem *ir_para* novos. Os terminais são I_1, I_2, I_5, I_7, I_8 e I_9. A Fig. 4.39 mostra os dez conjuntos de itens com seus *ir_para*. ∎

Forneceremos as regras pelas quais as funções sintáticas de ação e *ir_para* LR(1) são construídas a partir dos conjuntos de itens LR(1). As funções de ação e *ir_para* são representadas por uma tabela como antes. A única diferença está nos valores das entradas.

Algoritmo 4.10. Construção da tabela sintática LR canônica.

Entrada. Uma gramática aumentada G'.

Saída. As funções sintáticas canônicas LR *ação* e *ir_para* para G'.

Método.

1) Construir $C = \{I_0, I_1, \ldots, I_n\}$, a coleção de conjuntos de itens LR(1) para G'.

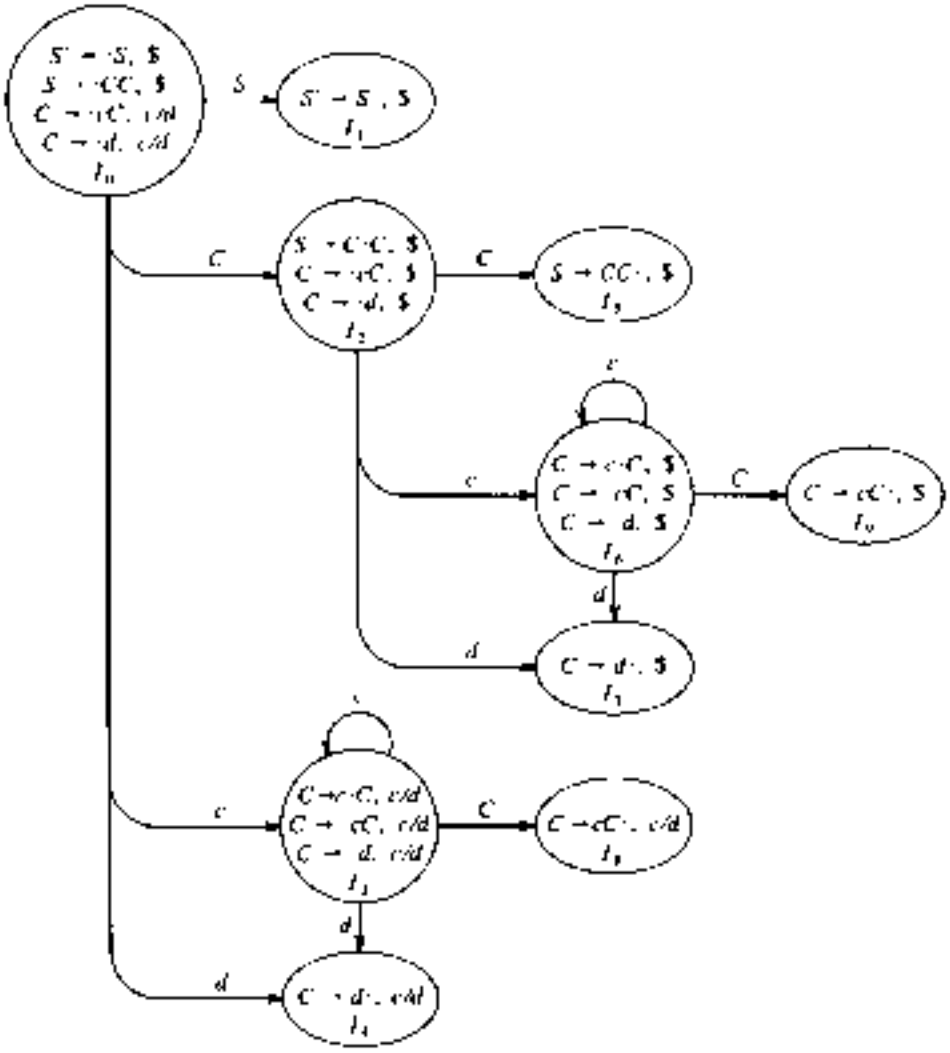


Fig. 4.39. O grafo de AFD para a gramática LR(1).

2 O restante do analisador sintático é construído a partir de I_0. As ações sintáticas para o estado s são determinadas como se segue:

a) Se $[X \to \alpha \cdot a\beta, b]$ estiver em I_s e $goto(I_s, a) = I_t$, então fazer $ação[s, a] = "empilhar\ t"$. Aqui, a é exigido ser um terminal.

b) Se $[X \to \alpha \cdot, a]$ estiver em I_s, então fazer $ação[s, a] = "reduzir\ X \to \alpha"$.

c) Se $[S' \to S \cdot, \$]$ estiver em I_s, então fazer $ação[s, \$] = "aceitar"$.

Se um conflito resultar das regras acima, a gramática não é considerada LR(1) e o algoritmo falha.

3 As transições devido para o estado s são determinadas como se segue: se $goto(I_s, A) = I_t$, então $devido[s, A] = t$.

4 Todas as entradas não definidas pelas regras 2 e 3 são tomadas como "erro".

5 O estado inicial do analisador sintático é aquele construído a partir do conjunto contendo o item $[S' \to \cdot S, \$]$.

A tabela é montada a partir das funções sintáticas de *ação* e de *devido*, produzidas pelo algoritmo 4.10, e dentro da tabela sintática LR(1) construída a partir do analisador sintático LR(1) canônico. Se as funções sintáticas de ações não possuírem entradas múltiplas definidas, então a gramática é chamada uma *gramática LR(1)*. Quaisquer conflitos restantes indicam que a gramática não pode ser analisada sintaticamente pelo método escolhido.

Exemplo 4.43. A tabela sintática canônica para a gramática 4.21 é mostrada na Fig. 4.40. As produções 1, 2 e 3 são $S \to CC$, $C \to cC$ e $C \to d$.

Cada gramática SLR(1) possui uma gramática LR(1), mas para uma gramática SLR(1) o analisador sintático LR pode ter menos estados

ESTADO	*ação*			*devido*	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

Fig. 4.40. Tabela sintática canônica para a gramática 4.21.

se ocorre estados duplicados. Os analisadores SLR para a mesma gramática. A gramática e os exemplos anteriores e SLR e possuem analisadores sintáticos SLR com sete estados, comparados com os nove da Fig. 4.40.

Construindo Tabelas Sintáticas LALR

Vamos agora descrever a segunda técnica de construção de tabelas sintáticas apresentada neste capítulo, a técnica LALR (look-ahead LR). Este método é frequentemente usado na prática porque as tabelas obtidas são consideravelmente menores do que as tabelas LR canônicas e, ao mesmo tempo, a maioria das construções sintáticas comuns das linguagens de programação pode ser expressa convenientemente por gramáticas LALR. O mesmo ocorre com tabelas para gramáticas SLR mas existem uma pequena quantidade de construções que não podem ser convenientemente tratadas por técnicas SLR (veja o Exemplo 4.54, por exemplo).

Para uma comparação do tamanho das tabelas, as tabelas SLR e LALR para uma gramática específica possuem o mesmo número de estados, e esse número é tipicamente de várias centenas para uma linguagem como Pascal. A tabela LR canônica teria tipicamente várias dezenas de estados para uma linguagem com o mesmo tamanho. Curiosamente, é possível construir tabelas SLR e LALR de que tabelas LR canônicas.

Como uma última introdução, vamos considerar de novo a gramática C (4.21), cujos conjuntos de itens LR(1) foram mostrados na Fig. 4.39. Também nos preocupamos com os estados aparentemente semelhantes tais como I_3 e I_6. Cada um desses estados possui itens com primeiros componentes iguais a $C \to d \cdot C$. Para I_3, os segundos componentes são todos c ou d, enquanto para I_6 todos são $\$$. Nenhum dos estados possui um conflito LR(1), mas se combinarmos I_3 com I_6, obtemos o conjunto de itens

$$C \to d \cdot C,\ c/d/\$$$
$$C \to \cdot dC,\ c/d/\$$$
$$C \to \cdot d,\ c/d/\$$$

para o qual ocorre um conflito redução/redução, uma vez que os itens $C \to d \cdot$ e $C \to \cdot d$ são ambos candidatos a redução com c ou d ou $\$$.

Estamos agora preparados para fornecer o passo-a-passo dos algoritmos de construção de tabela LALR. A idéia geral é fundir conjuntos de itens LR(1) que possuem primeiros componentes iguais, ou seja, juntar conjuntos com núcleos comuns. Em segundo, construímos uma tabela sintática a partir da coleção de conjuntos de itens combinados. O método que estamos prestes a descrever será primeiramente descrito em termos da gramática LALR de C. A construção de todos os conjuntos de itens LR(1) requer espaço e tempo demais para ser útil na prática.

Algoritmo 4.11. Uma construção fácil da tabela sintática LALR que algumas vezes requer muito espaço.

Entrada. Uma gramática aumentada G'.

Saída. As funções da tabela sintática LALR $action$ e $desvio$ para G'.

Método.

1. Construir $C = \{I_0, I_1, \ldots, I_n\}$, a coleção de conjuntos de itens LR(1) de G'.
2. Para cada núcleo que se apresente entre os conjuntos de itens LR(1), encontrar todos os conjuntos que tenham o mesmo núcleo e substituí-los por sua união.
3. Seja $C' = \{J_0, J_1, \ldots, J_m\}$ a coleção de conjuntos de itens LR(1) resultante. As ações sintáticas para o estado i são construídas a partir de J_i da mesma maneira que no algoritmo 4.10. Se existir um conflito de ação sintática, o algoritmo falha e produz um analisador sintático para a gramática fornecida (LALR(1)).
4. A tabela $desvio$ é construída como se segue. Se J for a união de um ou mais conjuntos de itens LR(1), isto é, $J = I_1 \cup I_2 \cup \cdots \cup I_k$, então os núcleos de $desvio(J, X)$ serão $desvio(I_1, X), desvio(I_2, X), \ldots, desvio(I_k, X)$. Visto que $I_1, I_2, \ldots, I_k$ possuem todos o mesmo núcleo, $desvio(I_1, X)$ terá o mesmo núcleo. Seja K a união de todos os conjuntos de itens que tenham o mesmo núcleo que $desvio(I_1, X)$. Então $desvio(J, X) = K$. ■

A tabela produzida pelo algoritmo 4.11 é chamada de *tabela sintática LALR* para G. Se não existirem conflitos de ações sintáticas, então a gramática é dita *LALR(1)*. A coleção de conjuntos de itens construída no passo (3) é chamada de *coleção LALR(1)*.

Exemplo 4.45. Consideremos outra vez a gramática 4.20, cujo grafo de dependência mostrado na Fig. 4.39. Como mencionamos, existem três pares conjuntos de itens que podem ser combinados. I_3 e I_6 são substituídos por sua união

$$\begin{aligned} I_{36}: \quad & C \to c \cdot C, c/d/\$ \\ & C \to \cdot cC, c/d/\$ \\ & C \to \cdot d, c/d/\$ \end{aligned}$$

I_4 e I_7 são substituídos por sua união

$$I_{47}: \quad C \to d \cdot, c/d/\$$$

e analogamente para I_8 e I_9:

$$I_{89}: \quad C \to cC \cdot, c/d/\$$$

As funções de ação e desvio LALR para os conjuntos convenientemente de itens são mostradas na Fig. 4.37.

Estado	ação			desvio	
	c	d	$\$$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Fig. 4.41. Tabela sintática LALR para a gramática (4.20).

Para ver por que os desvios são computados, consideremos o *desvio*(I_3, C). No conjunto original de itens LR(1), *desvio*$(I_3, C) = I_8$, e I_8 agora parte de I_{89}, e por conseguinte, fazemos *desvio*$(I_{36}, C) = I_{89}$. Poderíamos ter chegado a esta conclusão se considerássemos I_6, a outra parte de I_{36}. Isto é, *desvio*$(I_6, C) = I_9$, e I_9 agora parte de I_{89}. Para um outro exemplo, consideremos *desvio*(I_{36}, c), uma entrada que deve ser criada após a ação de empilhar de I_{36} a entrada c. Nos conjuntos originais de itens LR(1), *desvio*$(I_3, c) = I_6$. Uma vez que I_6 é agora parte de I_{36}, *desvio*(I_{36}, c) é também I_{36}. Consequentemente, a entrada na Fig. 4.41 para o estado 2 e entrada c é lida: empilhe o estado 36; significado: empilhe o símbolo de entrada c e o estado 36.

Quando apresentamos a uma cadeia proveniente da linguagem c^*dc^*d para o analisador sintático LR da Fig. 4.36 quanto o LALR da Fig. 4.41 realizam exatamente a mesma sequência de empilhamentos e reduções, apesar dos nomes dos estados na pilha poderem diferir, isto é, o analisador sintático LR coloca I_3 ou I_6 na pilha, o analisador sintático LALR na coloca I_{36}. Os relacionamentos vigem para uma gramática LALR em geral. Os analisadores sintáticos LR e LALR vão se portar exatamente para as cadeias corretas.

Erros, no entanto, são apresentados em momentos diferentes pelos analisadores sintáticos LR e LALR. Por exemplo, na entrada ccd seguida por $, o analisador sintático LR da Fig. 4.36 empilha I_3, I_3, I_6 e descobre um erro imediatamente, pois $ não é uma entrada válida no estado 6. O analisador LALR da Fig. 4.41 irá realizar os seguintes movimentos correspondentes, colocando

$$\text{D} \vdash 0\,c\,36\,c\,36\,d\,47$$

na pilha. Mas o estado 47 na entrada $ prevê a ação de reduzir através de $C \to d$. O analisador LALR irá corretamente substituir a pilha para

$$\text{D} \vdash 0\,c\,36\,c\,36\,C\,89$$

Agora a ação do estado 89 na entrada $ é reduzir através de $C \to cC$. A pilha se torna, agora,

$$\text{D} \vdash 0\,c\,36\,C\,89$$

cuja configuração também é válida, obtendo a pilha

$$0\,c\,2$$

Finalmente, o estado 2 prevê uma ação de erro na entrada $, sendo então o erro agora descoberto.

A Construção Eficiente de Tabelas Sintáticas LALR

Existem várias modificações que podemos fazer no algoritmo 4.11 para evitar a construção da coleção completa de conjuntos de itens LR(1) no processo de criação de uma tabela sintática LALR(1). A primeira observação é que podemos representar um conjunto de itens I pelo seu núcleo, isto é, por aqueles itens que ou são o item inicial $S' \to \cdot S, \$$ ou possuem o ponto em algum local que não o início de um lado direito.

Segundo, podemos computar as ações sintáticas geradas por I a partir do núcleo somente. Qualquer item que é criado por uma redução aparece em $A \to \gamma$ e esgota o corpo, a menos que $\gamma = \varepsilon$. A redução pela vez de $A \to \varepsilon$ especifica a entrada cujo somente será existir um item da forma $[B \to \alpha \cdot \gamma, b]$, tal que $C \Rightarrow^* Ab\gamma$ para algum γ e estiver em PRIMEIRO(γ). O conjunto de não terminais A tais que $C \Rightarrow^* Ab\gamma$ pode ser pré-computado para cada não terminal C.

As ações de empilhamento geradas por I podem ser determinadas a partir do núcleo de I como se segue. Empilhamos a entrada a se existir um item do núcleo $[B \to \alpha \cdot \beta, b]$ onde β começa com a, isto é, se $\beta \Rightarrow^* a\gamma$. O conjunto de tais a pode também ser pré-computado para cada B.

Aqui está como as transições de desvio para I podem ser computadas a partir do núcleo. Se $[B \to \alpha \cdot X\beta, b]$ estiver no núcleo de I, então $[B \to \alpha X \cdot \beta, b]$ estará no núcleo de *desvio*(I, X). O item $A \to X\beta, a]$ também estará no núcleo de *desvio*(I, X) se existir um item $[B \to \alpha \cdot A\gamma, b]$ de I e $C \overset{rm}{\Rightarrow} A\eta$ para algum η. Se pré-computarmos para cada par de não-terminais C e A se $C \Rightarrow^* A\eta$ para algum η, e computarmos os conjuntos de itens a partir dos itens iniciais, teremos um meio eficiente de realizar através dos conjuntos fechados de itens.

Para computar os conjuntos de itens LALR(1) de uma gramática aumentada G', começamos pelo núcleo $S' \to \cdot S$ do conjunto inicial de itens I_0. Em seguida, computamos os núcleos das transições de desvio a partir de I_0, como mencionado acima. Continuamos computando as transições de desvio para cada novo núcleo gerado até que tenhamos os núcleos de todas as coleções de itens LR(0).

Exemplo 4.46. Vamos considerar de novo a gramática aumentada

$$\begin{aligned} S' &\to S \\ S &\to L = R \mid R \\ L &\to * R \mid \mathbf{id} \\ R &\to L \end{aligned}$$

Os núcleos dos conjuntos de itens LR(0) para esta gramática são mostrados na Fig. 4.42.

Expandimos agora os núcleos introduzindo cada item LR(0) e os símbolos *lookahead* apropriados nos seguintes componentes. Para ver

I_0	$S' \to \cdot S$	I_7	$I \to I \cdot \mathbf{id}$
I_1	$S' \to S \cdot$	I_8	$S \to L \cdot = R$
I_2	$S \to L \cdot = R$ $R \to L \cdot$	I_9	$S \to L = R \cdot$
I_3	$S \to R \cdot$	I_{10}	$R \to L \cdot$
I_4	$L \to * \cdot R$		

Fig. 4.42. Núcleos dos conjuntos de itens LR(0) para a gramática (4.20).

apresentam *lookaheads* se propagam de um conjunto de itens I para o item $A \to \alpha \cdot X\beta$ em um conjunto LR(0) J se γA for o núcleo de J. Suponhamos que $I = \gamma$ e $A \to \alpha \cdot X\beta$ seja uma produção. Então o item LR(0) $A \to \alpha X \cdot \beta$ estará em $J = \gamma\alpha X$.

Suponhamos agora que estejamos computando os itens LR(1), assim como LR(0), e que $B \to \gamma \cdot A\delta, a$ esteja no conjunto I. Para cada valor de a (terminal ou $)$, $A \to \alpha \cdot X\beta, a$ estará em $J = \gamma\alpha X$, exatamente se $B \to \gamma A \cdot \delta, a$ estiver em $J = \gamma A$. Observe que a não tem de ser $\$$ ou $\mathbf{id}$; a pode ser qualquer terminal. Dizemos que $A \to \alpha \cdot X\beta, a$ em I gera o *lookahead* a para $A \to \alpha X \cdot \beta$ em $J = \gamma\alpha X$. Note também que a pode ser $)$ se $A = S'$ e $X = S$. Dizemos que $B \to \gamma A \cdot \delta, a$ em $J = \gamma A$ *propaga* o *lookahead* a para $A \to \alpha X \cdot \beta$ em $J = \gamma\alpha X$. Por definição, a é gerado espontaneamente para $A \to \alpha X \cdot \beta$ em $J = \gamma\alpha X$ se $A \to \alpha \cdot X\beta, a$ estiver em I e $X \neq \varepsilon$. Note também que a pode ser $)$ se $A \to \alpha X \cdot \beta$ estiver em $J = \gamma\alpha X$ e β for ε ou β for anulável.

Observe também que, se $A \to \alpha \cdot X\beta, a$ estiver em $I = \gamma$ e $X \Rightarrow^* \varepsilon$, então $A \to \alpha X \cdot \beta, a$ também estará em $J = \gamma\alpha X$. Dizemos, nesse caso, que o *lookahead* a se propaga de $B \to \gamma \cdot A\delta, b$ para $A \to \alpha X \cdot \beta, a$. Uma maneira simples para detectar quando um item LR(1) gera um *lookahead* espontaneamente, ou quando *lookaheads* se propagam, está contida no próximo algoritmo.

Algoritmo 4.62. Determinação dos *lookaheads*.

Entrada. O núcleo K de um conjunto de itens LR(0) I e um símbolo gramatical X.

Saída. Os *lookaheads* gerados espontaneamente pelos itens em I para os itens de núcleo em $GOTO(I, X)$ e os itens em I para os quais os *lookaheads* são propagados para os itens de núcleo em $GOTO(I, X)$.

Método. O algoritmo é fornecido na Fig. 4.63. É usado um símbolo fictício de *lookahead* $\#$ para detectar se os *lookaheads* são gerados espontaneamente ou propagados.

```
para cada item B → γ·δ em K faça início
    J := fecho({[B → γ·δ, #]});
    se [A → α·Xβ, a] estiver em J com a ≠ # então
        lookahead a é gerado espontaneamente para o item
            A → αX·β em GOTO(I, X)
    se [A → α·Xβ, #] estiver em J então
        os lookaheads se propagam de B → γ·δ em I para
            A → αX·β em GOTO(I, X)
fim
```

Fig. 4.63. Descobrindo *lookaheads* propagados e espontâneos.

Vamos agora considerar como fazer uso do algoritmo 4.62. Primeiro, sabemos que S' é um *lookahead* para $S' \to \cdot S$ no conjunto inicial de itens LR(0). O algoritmo 4.62 nos diz todos aqueles *lookaheads* gerados espontaneamente. Após isso, tudo o que precisamos fazer é propagar esses *lookaheads* para todos os lugares onde eles devam ir. Lembre-se de que sempre que um *lookahead* for gerado para um item LR(0) A, ele também deverá ser propagado para qualquer item LR(0) que tenha A como primeiro símbolo à direita do ponto. O procedimento geral é criar uma tabela para propagar *lookaheads* para todos os itens.

Exemplo. Uma gramática aumentada, G'.

Saída. Os núcleos de conjuntos de itens LALR(1) para G'.

Método.

1. Usando o método delineado acima, construa os núcleos dos conjuntos de itens LR(0) para G.
2. Aplicar o algoritmo 4.62 a cada núcleo de cada conjunto de itens LR(0) e símbolo gramatical X para determinar quais *lookaheads* são gerados espontaneamente para itens de núcleos em $GOTO(I, X)$, e quais itens em I são propagados *lookaheads* para itens de núcleos em $GOTO(I, X)$.
3. Inicialize uma tabela que forneça, para cada item de núcleo em cada conjunto de itens, os *lookaheads* associados. Inicialmente, cada item tem associados a si somente aqueles *lookaheads* que foram determinados no passo 2 acima como sendo gerados espontaneamente.
4. Realize repetidas passagens sobre os itens de núcleo em todos os conjuntos. Ao visitarmos um item, propagamos por meio de núcleo para quais itens ele propaga seus *lookaheads*, usando a informação tabulada em (3). Quando um certo de *lookaheads* para cada item de núcleo, não acrescentamos nenhum item a qualquer conjunto na próxima passagem sobre os itens de núcleo. Continuamos fazendo passagens sobre os itens de núcleo até que não possam ser propagados novos *lookaheads*.

Exemplo 4.47. Vamos construir os núcleos dos itens LALR(1) para a gramática (4.20) do exemplo 4.46. Os núcleos dos itens LR(0) foram mostrados na Fig. 4.42. Quando aplicamos o algoritmo 4.62 ao primeiro conjunto de itens I_0, computamos $GOTO(I_0, S)$, $GOTO(I_0, L)$, $GOTO(I_0, R)$, $GOTO(I_0, *)$ e $GOTO(I_0, \mathbf{id})$. Para $GOTO(I_0, S)$ temos

$$\begin{aligned} S' &\to S \cdot, \# \\ S &\to L \cdot = R, \# \\ S &\to R \cdot, \# \\ L &\to * \cdot R, = \\ L &\to \mathbf{id} \cdot, = \\ R &\to L \cdot, = \end{aligned}$$

Esses itens resultam do fecho provocado pelo gerador espontâneo dos *lookaheads*. O item $L \to * \cdot R, =$ faz com que $=$ seja colocado em $I_4 = GOTO(I_0, *)$ e também em $I_5 = GOTO(I_0, \mathbf{id})$. $L \to * \cdot R, =$ faz com que $=$ seja espontaneamente gerado para o item de núcleo $L \to * R \cdot$ em I_4, e $L \to \mathbf{id} \cdot, =$ faz com que $=$ seja espontaneamente gerado para o item de núcleo $L \to \mathbf{id} \cdot$ em I_5.

O padrão de propagação de *lookaheads* entre os itens de núcleo é determinado nos passos (2) e (3) do algoritmo 4.13. Está sumarizado na Fig. 4.44. Por exemplo, os dois itens de I_4 para os símbolos S, L, R, e $\mathbf{id}$ são respectivamente I_8, I_2, I_{12}, e I_5. Para I_8, computamos somente o fecho

I_0	$S' \to \cdot S$	I_1	$S' \to S \cdot, \#$
		I_2	$S \to L \cdot = R, \#$
		I_3	$S \to R \cdot, \#$
		I_4	$L \to * \cdot R, =$
		I_5	$L \to \mathbf{id} \cdot, =$
I_1	$S' \to S \cdot, \#$		
I_2	$S \to L \cdot = R, \#$ $R \to L \cdot, \#$	I_6	$S \to L = \cdot R, \#$
		I_7	$L \to * R \cdot, =$
I_3	$S \to R \cdot, \#$		
I_4	$L \to * \cdot R, =$	I_8	$R \to L \cdot, =$
		I_9	$L \to * \cdot R, \#$
		I_{10}	$L \to \mathbf{id} \cdot, \#$
I_5	$L \to \mathbf{id} \cdot, =$		
I_6	$S \to L = \cdot R, \#$	I_{11}	$R \to L \cdot, \#$
		I_8	$R \to L \cdot, =$
		I_{12}	$L \to * R \cdot, \#$
I_7	$L \to * R \cdot, =$		
I_8	$R \to L \cdot, =$		
I_9	$S \to L = \cdot R, \#$	I_{11}	$R \to L \cdot, \#$
		I_{10}	$L \to * \cdot R, \#$
		I_{12}	$L \to \mathbf{id} \cdot, \#$
I_{10}	$L \to \mathbf{id} \cdot, \#$		
I_{11}	$R \to L \cdot, \#$		
I_{12}	$L \to * R \cdot, \#$		

Fig. 4.44. Propagação de *lookaheads*.

CONJUNTO DE ITENS	FOLLOW			
	INICIO	PASSAGEM 1	PASSAGEM 2	PASSAGEM 3
I_0 $S' \to \cdot S$	$	$	$	$
I_1 $S' \to S \cdot$		$	$	$
I_2 $S \to L \cdot = R$		$	$	$
I_3 $R \to L \cdot$		$	$	$
I_4 $S \to R \cdot$		$	$	$
I_5 $L \to \cdot * R$	=	=/$	=/$	=/$
I_6 $L \to \cdot$ id	=	=/$	=/$	=/$
I_7 $S \to L = \cdot R$			$	$
I_8 $L \to * R \cdot$		=	=/$	=/$
I_9 $R \to L \cdot$		=	=/$	=/$
I_{10} $S \to L = R \cdot$				$

Fig. 4.45. Computação de FOLLOW.

exatamente um símbolo terminal de modo a seguir $[A \to \alpha \cdot a\beta]$. Consequentemente, V é $\{a\}$ para qualquer item nuclear em I_4 até I_9.

Na Fig. 4.45, mostramos os passos 2 e 3 do algoritmo 4.43. A coluna rotulada INÍCIO mostra os *followers* gerados espontaneamente para cada item nuclear. Na primeira passagem, o *follower* \$ se propaga a partir de $S \to L \cdot = R$ em I_2 para os seis itens listados na Fig. 4.44. O *follower* = se propaga a partir de $L \to *R$ em I_5 para os itens $L \to *R$ em I_5, $R \to L \cdot$ em I_3 e $R \to L \cdot$ em I_8. Também se propaga para $L \to$ id em I_6, mas esses *followers* já estão presentes. Na segunda e terceira passagens o único novo *follower* propagado é \$, descoberto para os sucessores de I_2 e I_4 na passagem 2 e para o sucessor de I_7 na passagem 3. Nenhum novo *follower* é propagado à passagem 4, e, por conseguinte, o conjunto final de *followers* encontrado na coluna mais à direita da Fig. 4.45.

Note-se que o resultado de empilhar redundante encontrado no Exemplo 4.39, ao se utilizar o método SLR, desaparece com esta técnica LALR. A razão está em que somente o *follower* \$ está associado a $R \to L \cdot$ em I_2, e consequentemente não há conflito com a ação sintática de empilhar com = gerada com base em $S \to L \cdot = R$ em I_2.

Compactação das Tabelas Sintáticas LR

Uma gramática típica de linguagem de programação com 50 a 100 terminais e 100 produções pode ter uma tabela sintática LALR com milhares de estados. A função de ação pode facilmente ter 20.000 entradas, cada uma requerendo pelo menos 8 bits para codificar. Claramente, uma codificação mais elaborada do que um vetor bidimensional pode ser importante. Iremos apontar três técnicas para explorar as características específicas das tabelas sintáticas LR.

Uma técnica para comprimir o campo de ação é reconhecer que usualmente muitas linhas da tabela de ação são idênticas. Por exemplo, na Fig. 4.40, os estados 0 e 3 possuem entradas idênticas de ação e, também, as entradas 2 e 6. Podemos, por conseguinte, guardar um espaço considerável, em um pequeno custo de tempo, se construirmos um apontador para cada estado num *array* unidimensional. Os apontadores dos estados comuns nessas situações apontarão para o mesmo local. Para se ter acesso a informações desse *array*, associamos a cada terminal um número de zero até um menos do que o número total de terminais, e usamos esse número como um deslocamento para a partir do valor do apontador de cada estado. Num dado estado, a ação sintática para o iésimo terminal será encontrada localizações além do valor do apontador para aquele estado.

Uma eficiência adicional pode ser atingida às expensas de um analisador sintático um tanto mais lento, geralmente considerado um negócio razoável uma vez que um analisador sintático LR consome somente uma pequena fração do tempo total de compilação. Em vez da criação da lista para as ações de cada estado. A lista consiste em pares (símbolo terminal, ação). A ação mais comum para um estado pode ser colocada no fim da lista em vez de um erro, pois o símbolo terminal poderá ser substituído pela notação "qualquer", significando que, se o símbolo de entrada não for encontrado até então na lista, deveremos realizar aquela ação, não importa o que a entrada seja. Sobrevindo as entradas de erro, elas podem ser substituídas seguramente por ações de redução, para uma futura inspeção ao longo de uma linha. Os erros serão detectados posteriormente, antes de um movimento para empilhar o símbolo de entrada.

Exemplo 4.48. Consideremos a tabela sintática da Fig. 4.31. Note-se primeiro que as ações para os estados 0, 4, 6 e 7 concordam. Podemos representá-las todas pela lista

SÍMBOLO	AÇÃO
id	s5
(	s4
qualquer	erro

O estado 1 possui uma lista com

+	s6
$	aceitar
qualquer	erro

No estado 2, podemos substituir as entradas de erro por r2, de tal forma que a redução através da produção 2 ocorrerá a qualquer entrada, menos +. Consequentemente, a lista para o estado 2 é

*	s7
qualquer	r2

O estado 3 possui somente ações de erro e r4. Podemos substituí-la pela entrada de tal forma que a lista para o estado 3 consista somente de um qualquer r4. Os estados 5, 10 e 11, também, são tratados similarmente. A lista para o estado 8 é

)	s15
+	s11
qualquer	erro

e para o estado 9:

)	s12
qualquer	r1

Podemos também codificar a tabela de ir_a através de uma lista, mas parece mais eficiente criar uma lista de pares para cada não-terminal A. Cada par da lista para A é da forma *estado_corrente proximo_estado*, tal que

$$ir\_a[estado\_corrente, A] = proximo\_estado$$

Esta técnica é útil porque nela tende a haver um número menor de estados em qualquer coluna da tabela de ir_a. A razão está em que eles são para os não-terminais, que somente ser um estado derivável a partir de um conjunto de itens no qual alguns itens possuem A imediatamente

imediatamente à esquerda de um ponto. Nenhum conjunto possui itens com X imediatamente à esquerda de um ponto se $X = \$$. Consequentemente cada estado aparece em uma coluna de *ir-para*.

Para maior redução de espaço, notamos que as entradas de erro na tabela de *ir-para* nunca são consultadas. Por conseguinte, podemos substituir cada entrada de erro pela entrada de não-erro mais comum naquela coluna. Essa entrada se torna o *default* a ser representado na lista para essa coluna, para qualquer símbolo gramatical no estado correspondente.

Exemplo 4.49. Consideremos a Fig. 4.31 de novo. A coluna para E' possui a entrada 10 para o estado 7 e todas as outras entradas no estado são erros. Podemos substituir esses por 3, o valor para a coluna E a lista

símbolo gramatical	próximo estado
$*$	10
qualquer	3

Novamente, notamos a lista especial para a coluna F:

id	8
qualquer	2

Para a coluna E podemos escolher um dos 3 para ser o *default*, dentre duas entradas não necessárias em cada caso. Por exemplo, poderíamos criar para a coluna E a lista

$($	6
qualquer	1

Se o leitor atualizar o número de entradas nas listas criadas neste exemplo, naturalmente adicionando operadores previamente omitidos, verá que as listas ação e *ir-para* determinadas para as listas de procedimento, não trazem impressionantes economias de espaço sobre a implementação de matriz da Fig. 4.31. No entanto, os ganhos de espaço sobre implementações de matriz podem ser enormes em exemplos mais realistas, por exemplo, em uma gramática para C. Infelizmente, a busca em listas é lenta, pois cada passo envolve um desvio de ponteiro e um teste de igualdade. Vemos também a representação sob a forma de matriz.

Devemos notar também que os métodos de compressão de tabelas para autômatos finitos que foram discutidos na Seção 3.9 podem também ser usados para representar tabelas sintáticas LR. A escolha desses métodos é discutida nos exercícios.

4.8 USANDO GRAMÁTICAS AMBÍGUAS

Com certeza não é função de qualquer gramática ambígua tal como ser LR e, evidentemente, não está em qualquer das classes discutidas na Seção 4.3. Certos tipos de gramáticas ambíguas, entretanto, são úteis na especificação e implementação de linguagens. Como veremos nesta seção, para construções de linguagens tais como expressões, uma gramática ambígua pode ser mais concisa e mais natural do que uma gramática não ambígua equivalente. Um outro uso de gramáticas ambíguas está em isolar construções sintáticas que ocorrem comumente para otimização de casos especiais. Com uma gramática ambígua podemos especificar as construções de casos especiais adicionando-as de forma explícita às produções da mesma.

Poderíamos enfatizar que apesar de gramáticas que usamos serem ambíguas, em todos os casos presentes especificamos regras de ambigüidade que permitem somente uma árvore gramatical para cada sentença. Desta forma, a especificação global da linguagem ainda se mantém não ambígua. Frisamos também que as construções ambíguas devem ser usadas com parcimônia e de forma a se manter cuidadosamente controlado. De outra forma não poderá haver garantia sobre que linguagem é reconhecida pelo analisador sintático.

Usando a Precedência e a Associatividade para Resolver Conflitos de Ações Sintáticas

Consideremos a expressão sobre linguagens de programação. A seguinte gramática para expressões aritméticas com os operadores $+$ e $*$

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id} \tag{4.29}$$

é ambígua, porque não especifica a associatividade nem a precedência dos operadores $+$ e $*$. A gramática não ambígua

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned} \tag{4.30}$$

gera a mesma linguagem mas confere $+$ uma precedência menor do que $*$ e torna ambos os operadores associativos à esquerda. Existem duas razões pelas quais descrevemos usar a gramática (4.29). Primeiro, como veremos, podemos facilmente modificar os níveis de associatividade e de precedência dos operadores $+$ e $*$ sem perturbar as produções de (4.29) ou o número de estados do analisador sintático resultante. Segundo, o analisador sintático para (4.29) irá gastar uma parte substancial de seu tempo reduzindo cadeias através das produções $E \rightarrow T$ e $T \rightarrow F$, cuja principal função é garantir a associatividade e precedência. O analisador sintático para a Fig. 4.22(1) não irá despender tempo reduzindo através dessas produções simples, como discutiremos.

I_0: $E' \rightarrow \cdot E$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot \mathbf{id}$

I_1: $E' \rightarrow E \cdot$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

I_2: $E \rightarrow (\cdot E)$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot \mathbf{id}$

I_3: $E \rightarrow \mathbf{id} \cdot$

I_4: $E \rightarrow E + \cdot E$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot \mathbf{id}$

I_5: $E \rightarrow E * \cdot E$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot \mathbf{id}$

I_6: $E \rightarrow (E \cdot)$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

I_7: $E \rightarrow E + E \cdot$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

I_8: $E \rightarrow E * E \cdot$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

I_9: $E \rightarrow (E) \cdot$

Fig. 4.46. Conjuntos canônicos LR(0) para a gramática aumentada (4.29).

Os conjuntos de itens LR(0) para (4.29) aumentada por $E' \rightarrow E$ são mostrados na Fig. 4.46. Uma vez que a gramática (4.29) é ambígua, os conflitos sintáticos serão gerados quando tentarmos produzir uma tabela sintática LR a partir dos conjuntos de itens. Os estados correspondentes aos conjuntos de itens I_7 e I_8 geram esses conflitos. Suponhamos que usemos a abordagem SLR para construir a tabela de ações sintáticas. O conflito gerado por I_7 entre redução através de $E \rightarrow E + E$ e empilhar $+$ ou $*$ não pode ser resolvido porque $+$ e $*$ estão, ambos, em SEGUINTE(E). Por conseguinte, ambas as ações deverão ser chamadas para as entradas $+$ e $*$. Um conflito similar é gerado em I_8, entre redução através de $E \rightarrow E * E$ e empilhar as entradas $+$ e $*$. De fato, cada um de nossos métodos de construção de tabelas sintáticas irá gerar esses conflitos.

Entretanto, esses problemas podem ser resolvidos usando-se as informações de precedência e associatividade para $+$ e $*$. Consideremos a entrada **id** $+$ **id** $*$ **id**, que faz com que o analisador sintático baseado na Fig. 4.46 entre no estado 7 após processar id + id, em particular, o analisador sintático empilha a configuração

Pilha	Entrada
$0\,E_1 \quad 1\,/\,7$	**id** $)$

Assumindo que o leitor possua algum conhecimento sobre o funcionamento de um analisador sintático empilhador, vamos explicar o funcionamento desse analisador sintático para a expressão. Isto é, vamos analisar a análise SLR da Fig. 4.31, para a mesma linguagem. Lembre-se de que um analisador sintático de precedência de operadores faz. Por outro lado, sabemos que * tem precedência sobre +. Sabemos que o analisador sintático deve reduzir $E + E$ para E. Por conseguinte, a precedência relativa de * sobre + faz com que uma configuração sintática contendo $E + E * E$ empilhado resulte em $E + E$ sendo reduzido a E, e não em uma redução de $E * E$ a E. Vamos ver como essas ações sintáticas são executadas pelo analisador SLR e de precedência de operadores, tanto para a gramática do Exemplo 4.34.

Em nosso exemplo, assumindo que + seja associativo à esquerda, a ação correta do estado 2 com entrada + deveria ser a redução através de $E \to E + E$. Assumindo que + tenha precedência sobre *, a ação do estado 7 também deveria ser a mesma. Surpreendentemente, assumindo que + seja associativo à esquerda, tenha a precedência sobre * e também que o estado 8, que pode aparecer no topo da pilha somente quando $E * E$ é o conteúdo desses três símbolos gramaticais do topo, deveria ser a ação de reduzir $E + E * E$ a E, também a ação correta quando * tem precedência sobre +. Só nesse caso a gramática é ambígua. Vamos ver como o analisador sintático SLR e o analisador de precedência de operadores tratam cada um dos estados, quando a entrada for +, e também quando for *, e assumindo que + seja associativo à esquerda.

Procedendo dessa forma, obtemos a tabela sintática LR mostrada na Fig. 4.47. As produções 1 a 3 são $E \to E + E$, $E \to E * E$ e $E \to$ **id**, respectivamente. É interessante que a tabela de ações sintáticas SLR seja preenchida pelo uso das regras de precedência e associatividade de operadores usuais para a gramática da Fig. 4.31, mostrada na Fig. 4.31. As gramáticas ambíguas muitas vezes podem ser tratadas desta forma, mas é difícil entender o funcionamento de analisadores sintáticos LALR ou LR canônicos.

Estado	ação					ir_a	
	id	+	*	(	)	E	$\$$
0	s4			s3		1	
1		s5	s6				acc
2	s4			s3		7	
3		r4	r4		r4		
4	s4			s3		8	
5	s4			s3		9	
6		s4	s6		s6		
7		r1	s6		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Fig. 4.47. Tabela sintática para a gramática da Fig. 4.22.

A Ambigüidade do "*Else*-Vazio"

Consideremos de novo a seguinte gramática para comandos condicionais:

$$\begin{aligned} cmd &\to \textbf{if}\ expr\ \textbf{then}\ cmd\ \textbf{else}\ cmd \\ &\mid \textbf{if}\ expr\ \textbf{then}\ cmd \\ &\mid outro \end{aligned}$$

Como mencionamos na Seção 4.3, esta gramática é ambígua porque não resolve a ambigüidade do *else*-vazio. Para simplificar a discussão, vamos considerar uma construção da gramática acima, onde o lugar de **if** *expr* **then** e de **else** é ocupado por símbolos terminais individuais. Podemos então estudar a gramática, com a produção adicional $S' \to S$:

$$\begin{aligned} S' &\to S \\ S &\to i\,S\,t\,S \mid i\,S\,t\,S\,e\,S \mid a \end{aligned} \qquad (4.24)$$

Os conjuntos de itens LR(0) para a gramática (4.24) são mostrados na Fig. 4.48. A ambigüidade surge porque a ação sintática correta em I_4 sob a entrada e não é determinada. Seja $S \to i\,S\,t\,S$ o primeiro item em I_4, e seja $S \to i\,S\,t\,S\,e\,S$ o segundo item em I_4. Se e for a entrada, podemos empilhar e e ir para o estado 5, ou podemos reduzir pelo primeiro item. Qual ação é a correta? Se e for o símbolo de entrada, devemos empilhar, porque **else** está sendo empilhado com relação a **then**, e não **then** com relação a **else**, porque estamos tentando formar o lado direito que começa com *iSeS*, no topo da pilha. Se o que se segue a e na entrada não puder ser reduzido a eS, então podemos reduzir iS a S em pelo menos um caso (veja I_5), e talvez mais. Uma análise mais detalhada mostra que, para essa gramática, sempre é correto empilhar.

Se os leitores estiverem familiarizados com o funcionamento de um empilhador redutor em LR, deverão ser capazes de construir uma tabela sintática SLR, assumindo a partir dos conjuntos de itens da Fig. 4.48, usando-se a solução para o conflito de ações sintáticas em I_4 sob a entrada e mostrada na Fig. 4.49. As produções 1 a 3 são $S \to i\,S\,t\,S$, $S \to i\,S\,t\,S\,e\,S$ e $S \to a$, respectivamente.

Por exemplo, em relação aos estados dos símbolos gramaticais mostrados na Fig. 4.50, correspondentes a $cmd_1\ e\ cmd_2$ como

$$\begin{aligned} I_0:\ & S' \to \cdot S \\ & S \to \cdot i\,S\,t\,S \\ & S \to \cdot i\,S\,t\,S\,e\,S \\ & S \to \cdot a \\ I_1:\ & S' \to S \cdot \\ I_2:\ & S \to i \cdot S\,t\,S \\ & S \to i \cdot S\,t\,S\,e\,S \\ & S \to i \cdot S \\ & S \to i \cdot a \\ I_3:\ & S \to a \cdot \\ I_4:\ & S \to i\,S \cdot t\,S \\ & S \to i\,S \cdot t\,S\,e\,S \\ I_5:\ & S \to i\,S\,t \cdot S \\ & S \to i\,S\,t \cdot S\,e\,S \\ & S \to i \cdot S \\ & S \to \cdot a \\ I_6:\ & S \to i\,S\,t\,S \cdot \\ & S \to i\,S\,t\,S \cdot e\,S \\ I_7:\ & S \to i\,S\,t\,S\,e \cdot S \\ & S \to \cdot i\,S\,t\,S \\ & S \to \cdot i\,S\,t\,S\,e\,S \\ & S \to \cdot a \\ I_8:\ & S \to i\,S\,t\,S\,e\,S \cdot \end{aligned}$$

Fig. 4.48. Conjuntos LR(0) para a gramática (4.24).

Estado	ação				ir_a
	i	t	e	$\$$	S
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4		s5		r2	
5	s2		s3		6
6		r1		r1	

Fig. 4.49. Tabela sintática LR para a gramática ambígua do *else*-vazio.

	Pilha	Entrada
(1)	0	$a_1 a_2 a_3$
(2)	0 a_1 2	$a_2 a_3$
(3)	0 a_1 2 a_2 3	a_3
(4)	0 a_1 2 a_2 3 a_3 4	
(5)	0 a_1 2 a_2 3 a_3 4	
(6)	0 a_1 2 a_2 3 a_3 4 5	
(7)	0 a_1 2 a_2 3 a_3 4 5 3	
(8)	0 a_1 2 a_2 3 a_3 4 5 3	
(9)	0 a_1 2 a_2 3 4	
(10)	0 a_1 1	

Fig. 4.50 Ações analíticas ascendentes para a entrada $a_1 a_2 a_3$

...dores vazios. A tabela SLR resultante é exatamente a mesma que a mostrada na Fig. 4.20, mas agora as ações conflitantes são resolvidas pela relação de precedência de $\cdot$ e $\cdot$ com $\$$ à direita.

Uma vantagem de tais especificações compactas de analisadores sintáticos ascendentes para expressões aritméticas é que a gramática não precisa ser modificada de nenhuma forma para implementar a precedência. Veremos mais adiante que a mesma técnica pode ser usada para analisadores LR, mas que as tabelas de ações são muito menores se a gramática for ambígua.

Ambigüidades para Produções Tipo Caso Especial

Nesta seção final, sugerimos a utilidade das gramáticas ambíguas ao se introduzirmos uma produção ambígua para especificar um caso especial de uma construção sintática geralmente mais geral pelo resto da gramática. Ao adicionarmos a produção extra, geramos o conflito de ações sintáticas. Podemos frequentemente resolver o conflito sustentadamente através de uma escolha de maneira adequada para reduzir cada uma das produções do caso especial. A ação correta para cada produção depende de como o caso especial deve ser tratado pela tradução.

Uma aplicação dessas ambigüidades de caso especial foi feita por Kernighan e Cherry [1975] em um pré-processador de composição de equações EQN, que foi usado para auxiliar a composição do original desta tradução. Em EQN, a sintaxe de uma expressão matemática é descrita por uma gramática que inclui operadores de subscritos sub e um operador de superscritos sup, como mostrado no fragmento de gramática (4.250). Assim, expressões como x_{i+1} podem ser descritas por expressões compostas em EQN, como x sub i sup 2 sup {i+1}. Chamamos sub e sup de operadores de caixa.

$$
\begin{array}{lll}
(1) & E & \rightarrow E \text{ sub } E \text{ sup } E \\
(2) & E & \rightarrow E \text{ sub } E \\
(3) & E & \rightarrow E \text{ sup } E \\
(4) & E & \rightarrow \{ E \} \\
(5) & E & \rightarrow c
\end{array}
\tag{4.250}
$$

A gramática (4.250) é ambígua, pois as operações sub e sup não são associativas à esquerda nem à direita, e também porque sub tem precedência igual à de sup. Mesmo que estas ambigüidades sejam eliminadas, como na associatividade e precedência de + e *, ainda temos a sentença E sub E sup E sub E sup E com duas derivações à esquerda e duas à direita, e a gramática não seria ambígua. Isto porque a produção (1) é de um caso especial de produções geradas pelas produções (2) e (3), especificamente as produções da forma E sub E_1 sup E_2. A razão para se tratar de maneira especial expressões com essa combinação de operadores é que muitos compositores de tipos preferem associar uma expressão como x sub i sup 2 como $(x_i)^2$ e não como x_{i^2}. Adicionando-se explicitamente uma produção tipo caso especial, Kernighan e Cherry salvaram a EQN a produção mais caso especial de todas.

Para ver como este tipo de ambigüidade pode ser tratado em analisadores LR, vamos construir uma tradução sintática SLR para a gramática (4.250). Os conjuntos de itens LR(0) para esta gramática são mostrados na Fig. 4.51. Nesta coleção, os conjuntos de itens I_7, I_8 e I_9 geram conflitos de ações sintáticas. I_7, I_8 e I_9 geram conflitos de empilhar/reduzir nos símbolos sub e sup porque a associatividade e precedência desses operadores não são especificadas. Resolvendo-se esses conflitos de ações sintáticas quando fazemos sub e sup associativos à direita e com sub tendo precedência mais alta que sup, obtemos as ações desejadas para o caso especial. Por consequinte, empilhar é a ação correta em cada caso.

I_{10} também gera um conflito redução/redução, no caso das entradas } e $\$$, entre as duas produções

$$
\begin{array}{lll}
E & \rightarrow E \text{ sub } E \text{ sup } E \\
E & \rightarrow E \text{ sup } E
\end{array}
$$

O estado I_{10} está à esquerda do símbolo sub e quando tais comandos são seguidos por uma entrada que pode estar no lado direito de E sub E sup na pilha. Se resolvermos o conflito fazendo/fazendo com que a produção (1) estenda mais tarde uma expressão da forma E sub E sup E como um caso especial. Usando-se as regras de ambigüidade obtemos a tabela sintática SLR mostrada na Fig. 4.52.

Escrever gramáticas ambíguas que tolerem casos especiais de convenções sintáticas é muito útil. Para ajudar a evitar qualquer dificuldade, o leitor é encorajado a construir uma gramática não ambígua equivalente para (4.250) que trate de expressões da forma E sub E sup E.

Recuperação de Erros na Análise Sintática LR

Um analisador sintático LR irá detectar um erro assim que for impossível continuar a análise para qualquer prefixo de cadeia de entrada, mesmo que ele não tenha lido nenhum símbolo além do prefixo. Os erros nunca são

I_0: $E' \rightarrow \cdot E$
$E \rightarrow \cdot E$ sub E sup E
$E \rightarrow \cdot E$ sub E
$E \rightarrow \cdot E$ sup E
$E \rightarrow \cdot \{ E \}$
$E \rightarrow \cdot c$

I_1: $E' \rightarrow E \cdot$
$E \rightarrow E \cdot$ sub E sup E
$E \rightarrow E \cdot$ sub E
$E \rightarrow E \cdot$ sup E

I_2: $E \rightarrow \{ \cdot E \}$
$E \rightarrow \cdot E$ sub E sup E
$E \rightarrow \cdot E$ sub E
$E \rightarrow \cdot E$ sup E
$E \rightarrow \cdot \{ E \}$
$E \rightarrow \cdot c$

I_3: $E \rightarrow c \cdot$

I_4: $E \rightarrow E$ sub $\cdot E$ sup E
$E \rightarrow E$ sub $\cdot E$
$E \rightarrow \cdot E$ sub E sup E
$E \rightarrow \cdot E$ sub E
$E \rightarrow \cdot E$ sup E
$E \rightarrow \cdot \{ E \}$
$E \rightarrow \cdot c$

I_5: $E \rightarrow E$ sup $\cdot E$
$E \rightarrow \cdot E$ sub E sup E
$E \rightarrow \cdot E$ sub E
$E \rightarrow \cdot E$ sup E
$E \rightarrow \cdot \{ E \}$
$E \rightarrow \cdot c$

I_6: $E \rightarrow E$ sub $E \cdot$ sup E
$E \rightarrow E \cdot$ sub E
$E \rightarrow E \cdot$ sup E
$E \rightarrow \{ E \cdot \}$

I_7: $E \rightarrow E$ sub E sup $E \cdot$
$E \rightarrow E \cdot$ sub E sup E
$E \rightarrow E \cdot$ sub E
$E \rightarrow E \cdot$ sup E

I_8: $E \rightarrow E$ sub $E \cdot$ sup E
$E \rightarrow E \cdot$ sub E
$E \rightarrow E \cdot$ sup E

I_9: $E \rightarrow E$ sup $E \cdot$
$E \rightarrow E \cdot$ sub E sup E
$E \rightarrow E \cdot$ sub E
$E \rightarrow E \cdot$ sup E

I_{10}: $E \rightarrow E$ sub E sup $E \cdot$
$E \rightarrow E \cdot$ sub E sup E
$E \rightarrow E \cdot$ sub E
$E \rightarrow E \cdot$ sup E

I_{11}: $E \rightarrow \{ E \cdot \}$

Fig. 4.51 Conjuntos de itens LR(0) para a gramática (4.250)

Estado	ação						desvio
	sub	sup	(	)	,	$	L
0			s2		s3		1
1	s4	s5				ac	
2			s2		s3		6
3	r5	r5		r5		r5	
4			s2		s3		7
5			s2		s3		8
6	s4	s5		s9			
7	s4	s10		r2		r2	
8	s4	s5		r3		r3	
9	r4	r4		r4		r4	
10			s2		s3		11
11	s4	s5		r1		r1	

Fig. 4.52. Tabela sintática para a gramática (4.25)

detectados pelo conjunto da tabela de desvio. Diferentemente de um analisador sintático de precedência de operadores, um analisador LR não comete erros tão logo não exista continuação válida para a porção da entrada processada até então. Em outras palavras, um analisador LR jamais irá realizar uma ação redundante, antes de anunciar um erro. Os analisadores sintáticos SLR e LR podem realizar reduções adicionais quando de erros, mas jamais irão empilhar um símbolo incorreto de entrada.

No analisador sintático LR, podemos implementar a recuperação na modalidade desespero assim segue. Percorremos a pilha até que o estado contenha desvio para um símbolo não-terminal particular A sendo encontrado. Zero ou mais símbolos de entrada são descartados até que seja encontrado um símbolo a que possa legitimamente se seguir a A. O analisador sintático empilha então o estado $desvio[s, A]$ e continua sua análise sintática normal. Pode haver mais de uma escolha para o não-terminal A. Normalmente, essas serão não terminais representando estruturas importantes do programa, tais como expressões, enunciados ou blocos. Por exemplo, se A fosse uma expressão (normal), a poderia ser um terminador ou um ")".

Este método de recuperação retira da pilha toda a base que contém o erro sintático. O analisador sintático determina que uma cadeia derivável a partir de A contém um erro. Parte dessa cadeia foi processada, o resultado sendo uma sequência de estados no topo da pilha. O restante da cadeia ainda está na entrada e o analisador testa pular sobre o resto dessa cadeia, procurando por um símbolo de entrada que possa legitimamente se seguir a A. Através da remoção dos estados que estão na pilha, pulando sobre a entrada e empilhando $desvio[s, A]$, o analisador sintático pretende ter encontrado uma maneira de A ocorrer sobre a qual se está tentando analisar a entrada.

A recuperação em nível de frase é implementada através de examinar cada entrada de erro na tabela sintática LR e decidir sobre uma base do uso da linguagem, que erro do programador é mais propenso a dar origem àquele erro detectado. Um procedimento apropriado de recuperação pode então ser construído, presumivelmente o topo da pilha e/ou os próximos símbolos de entrada serão modificados numa forma considerada apropriada à cada entrada de erro.

Comparado com analisadores sintáticos de precedência de operadores, o projeto de rotinas específicas de tratamento de erro para um analisador sintático LR é relativamente fácil. Um particular, não temos que nos preocupar com as reduções equivocadas: qualquer ação pedida pela tabela por um analisador sintático LR está certamente correta. Podemos então preencher cada entrada em branco no campo ação com um apontador para uma rotina de erro que realiza ações apropriadas selecionadas pelo projetista do compilador. As ações devem incluir a inserção ou remoção de símbolos da pilha (ou possivelmente de outras estruturas) e a alteração ou transposição de símbolos da entrada exatamente como para um analisador sintático de precedência de operadores. Como aquele caso isolado, passagens futuras devem ser escolhidas sem permitir a possibilidade de um analisador sintático LR entrar num laço infinito. Uma estratégia particularmente segura é garantir que pelo menos um símbolo de entrada será removido ou eventualmente empilhado ou que a pilha será eventualmente reduzida se o fim da entrada tiver sido atingido. Uma técnica para esse propósito é descrita ao final da seção.

Exemplo 4.50. Considere novamente a gramática de expressões

$$E \to E + T \mid T \qquad T \to T * F \mid F \qquad F \to (E) \mid \mathbf{id}$$

A Fig. 4.53 mostra a tabela sintática LR da Fig. 4.42 (p. 213), com rotinas de erro incluídas para a detecção e as operações de erros. Modificamos cada estado que chama por uma redução particular em algum símbolo de entrada, substituindo as entradas de erro naquele estado pela redução. Essa modificação possui o efeito de postergar a detecção de erros até quando algumas reduções sejam feitas, mas o erro ainda será capturado antes que qualquer outro movimento de empilhar. As entradas de erro são mensagens prováveis da Fig. 4.42, todas sendo chamadas para rotinas de erro.

Estado	id	(	)	+	*	$	desvio
							E	T	F
0	s5	e1	e1	e2	e2	e1	1	2	3
1	e3	s4	e5	s6	e2	acc			
2	e4	e1	e1	e2	e2	e1			
3	r4	r4	r4	r4	r4	r4			
4	s5	e1	e1	s7	e2	e1	7	2	3
5	e3	e1	e1	e2	e2	e1			
6	s5	e1	e1	s7	e2	e1		8	3
7	e3	s4	s9	e3	s9	e2			
8	r1	r1	e5	r1	r1	r1			
9	r2	r2	r2	r2	r2	r2			
10	r3	r3	r3	r3	r3	r3			

Fig. 4.53. Tabela sintática LR com rotinas de erro

As rotinas de erro são como segue. A semântica dessas ações, como os erros que elas representam para as ações de erro do Exemplo 4.42 (precedência de operadores) deveria ser evidente. Empregando o analisador sintático LR e frequentemente manipulado pelo processador de ações do analisador sintático de precedência de operadores.

e1: // Esta rotina é chamada a partir dos estados 0, 2, 4, 5, 6, 7 e 8, todos estados que esperam pelo início de um operando, ou **id** ou um parêntese à esquerda. Em ligação com o operador — assim que o início da entrada for encontrado —
empilhar um **id** imaginário e cobrir o estado 5 (o desvio de todos os estados 0, 2, 4, 6, 7 e 8 para **id**)
e2: // Esta rotina é chamada a partir dos estados 0, 1, 2, 4, 6, 7 e 8, todos encontrados um parêntese à direita. //
remover os parênteses à direita a partir da entrada
emitir o diagnóstico "parênteses à direita não balanceados"
e3: // Esta rotina é chamada a partir dos estados 1 ou 6 todos os quais esperam um operador ou um fim de parênteses à direita. //
empilhar + e cobrir (o mesmo estado 4)
emitir o diagnóstico "operador ausente"
e4: // Esta rotina é chamada a partir do estado 6 quando o fim da entrada é antecipado. O estado 6 espera um operador ou parênteses à direita. //
empilhar um parênteses à direita e cobrir com o estado 9.
emitir o diagnóstico "parênteses à direita ausente"

Ao se receber a entrada incorreta **id** +) seguido de fim de entrada (Exemplo 4.42), a sequência de configurações alcançadas pelo analisador sintático é mostrada na Fig. 4.54.

Se você é paranoico, você pode querer verificar que a pilha não cresce sem limites durante uma sequência de inserções de erros. Cada rotina de erro deve ter a propriedade que ou ela remove um símbolo de entrada, ou ela empilha um estado que não aparece na mesma célula da tabela sintática, ou ela reduz a pilha.

Estado	Entrada	Mensagens e Ações de Erro
0	id + id $	
0id5	+ id $	
0F3	+ id $	
0T2 + 4	id $	
0T2 + 4id5	$	[illegible]
		e2 empilha parêntese à direita
0T2 + 4id5	$	[illegible]
		e3 empilha id 5
0T2 + 4F3	$	
0T2	$	

Fig. 4.54. Ações de erro para o exemplo de recuperação de erros feitas por um analisador sintático LR.

4.9 GERADORES DE ANALISADORES SINTÁTICOS

Esta seção mostra como um gerador de analisadores sintáticos pode ser usado para facilitar a construção da vanguarda de um compilador. Iremos usar o gerador de analisadores sintáticos LALR Yacc como base de nossa discussão, uma vez que o mesmo implementa muitos dos conceitos discutidos nas duas seções anteriores e está amplamente disponível. Yacc significa "mais um compilador de compiladores", refletindo a popularidade dos geradores de analisadores sintáticos no início dos anos 70. A primeira versão do Yacc foi criada por S. C. Johnson. Yacc está disponível como um comando do sistema UNIX e tem sido usado para auxiliar a implementação de centenas de compiladores.

O Gerador de Analisadores Sintáticos Yacc

Um tradutor pode ser construído utilizando-se o Yacc da forma descrita na Fig. 4.55. Primeiro, um arquivo, digamos, [illegible]

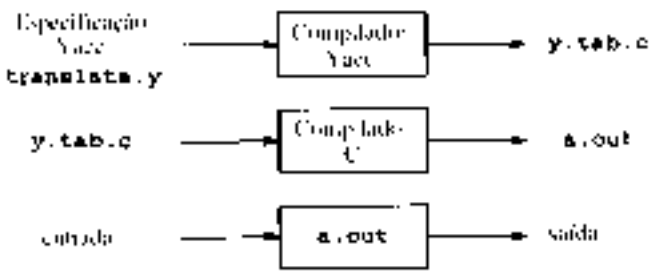


Fig. 4.55. Criação de um tradutor de entrada-saída através do Yacc.

contendo uma especificação Yacc do tradutor é preparado. O comando de sistema UNIX

[illegible]

cria o programa em C [illegible], um programa C chamado [illegible] usando o método LALR discutido na Seção 4.7. O programa [illegible] é uma representação de um analisador sintático LALR escrito em C, juntamente com outras rotinas C que o usuário poderia ter preparado. A tabela sintática LALR é compactada como descrito na Seção 4.7. Através da compilação de [illegible] juntamente com a biblioteca [illegible] que contém o programa de análise sintática LR, usando o comando

[illegible]

obtemos o programa objeto desejado, [illegible], que realiza a tradução especificada pelo programa Yacc original. Se outros procedimentos forem necessários, podem ser compilados ou carregados com [illegible] exatamente como em qualquer programa C.

Um programa Yacc possui três partes:

[illegible]

Exemplo 4.51. Para ilustrar o preparo de um programa Yacc, vamos construir uma calculadora de mesa simples que lerá uma expressão aritmética, a avaliará e imprimirá seu valor numérico. Iremos construir a calculadora de mesa com a seguinte gramática para expressões aritméticas:

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \textbf{dígito} \end{aligned}$$

O token **dígito** é um único dígito entre 0 e 9. Um programa Yacc para a calculadora de mesa derivada desta gramática é mostrado na Fig. 4.56. ❑

[illegible]

Fig. 4.56. Especificação Yacc de uma calculadora de mesa simples.

A parte de declarações. Existem duas seções opcionais na parte de declarações de um programa Yacc. Na primeira, colocamos declarações C ordinárias, delimitadas por %{ e %}. Aqui colocamos declarações de quaisquer variáveis temporárias usadas pelas regras de tradução das segunda e terceira seções. Na Fig. 4.56, esta seção contém somente o comando

[illegible]

que causa o pré-processador C incluir o arquivo de cabeçalho padrão [illegible] que contém o predicado [illegible]

[illegible]

Transcrição parcial de declarações extraídas da gramática Na página 450, o comando

declara [illegible] como tokens. Os tokens declarados nessa seção podem ser usados nas seções de regras da especificação Yacc

Após as regras da gramática. Na parte de especificação Yacc após o primeiro par [illegible] colocamos as regras de tradução. Cada regra consiste numa produção gramatical e da ação semântica associada. Um conjunto de produções que anteriormente escrevemos como

[illegible]

seria escrito em Yacc como

[illegible]

Numa produção Yacc, os caracteres simples entre aspas simples (como '+' e '(') são considerados terminais, e cadeias de letras e dígitos sem aspas representam não terminais ou tokens, dependendo se eles aparecem ou não como cabeças de produções. Ações semânticas podem ser especificadas por blocos de instruções em pseudo-código, cercadas por chaves { e }. Nas ações semânticas, o símbolo $$ é usado para referenciar o valor semântico associado ao símbolo à esquerda, e $1, $2, ... referem-se aos valores associados aos símbolos à direita. Ações semânticas normalmente ocorrem no fim da produção, após todos os componentes terem sido analisados. Ações semânticas podem também ser colocadas no meio de produções, caso em que são executadas assim que o analisador sintático reconhece a parte da produção à esquerda da ação. Ações semânticas podem ser usadas para impor restrições sensíveis ao contexto que não podem ser especificadas na gramática livre de contexto, ou para gerar código intermediário.

Uma ação semântica Yacc é uma sequência de comandos em C. Normalmente a ação semântica é simplesmente uma expressão que define o valor associado ao não-terminal à esquerda da produção. Por exemplo, uma ação para a produção $E \rightarrow E + T$ pode ser $$ = $1 + $3;. Lembre-se que $1 e $3 referem-se aos valores associados aos dois símbolos à direita, E e T, respectivamente. O valor associado ao símbolo à esquerda é definido como $$. Ações semânticas podem ser mais complexas do que simples regras de avaliação. Por exemplo, é comum a construção de árvores sintáticas, onde cada nó é representado por uma estrutura ou registro alocado dinamicamente. Ponteiros para nós filhos são normalmente armazenados nos campos do nó pai, e ponteiros para nós podem ser passados como valores semânticos. O parágrafo a seguir ilustra esse estilo de programação.

Uma ação semântica Yacc é uma sequência de comandos em C. Normalmente a ação semântica é simplesmente uma expressão que define o valor associado ao não-terminal à esquerda da produção. Por exemplo, uma ação para a produção $E \rightarrow E + T$ pode ser $$ = $1 + $3;. Lembre-se que $1 e $3 referem-se aos valores associados aos dois símbolos à direita, E e T, respectivamente. O valor associado ao símbolo à esquerda é definido como $$. Ações semânticas podem ser mais complexas do que simples regras de avaliação. Por exemplo, é comum a construção de árvores sintáticas, onde cada nó é representado por uma estrutura ou registro alocado dinamicamente. Ponteiros para nós filhos são normalmente armazenados nos campos do nó pai, e ponteiros para nós podem ser passados como valores semânticos. O parágrafo a seguir ilustra esse estilo de programação.

Na especificação Yacc a seguir, as duas produções $E \rightarrow E + T$ e $E \rightarrow T$ são implementadas por uma função `int nptr(int, int, int)` e a produção $T \rightarrow (E)$ é implementada pela função `int pnptr(int, int, int)`. Ambas as funções criam um nó em uma árvore sintática e retornam um ponteiro para o nó criado. Nas especificações Yacc, é preciso definir o tipo dos valores semânticos. Isso é feito através da definição `%union`, que é copiada para o arquivo de saída gerado pelo Yacc. Nesse exemplo, definimos que os valores semânticos são inteiros.

[illegible]

Note que o não-terminal termo na gramática acima, e os terminais no lado gramatical da direita, empregam que $1 e $3 se referem a um valor inteiro associado a um ponteiro para nó. Isso é feito através da definição `%type <ival>` na segunda parte da especificação Yacc, onde `ival` é o nome do campo inteiro da união definida em `%union`. Os tokens `num` e `id` também recebem valores inteiros, que são ponteiros para nós folha. Ações semânticas *default* para produções com um único símbolo à direita são da forma `$$ = $1;`. Em geral, a ação semântica *default* é `$$ = $1; $2 = $2; ... $n = $n;`.

Note que ações semânticas podem ocorrer no meio de produções, como em

[illegible]

na especificação Yacc. A produção diz que uma variável para a cadeia de declarações deve ser uma expressão seguida por um caractere de nova linha. A ação semântica associada a essa produção imprime o valor da expressão seguida por um caractere de nova linha.

A partir dos exemplos fornecidos, vê-se como é fácil construir especificações Yacc simples usando tokens. Uma limitação leva a [illegible] precisa ser provida pelo usuário. Outros pontos importantes, tais como formas de recuperação de erros, também são tratados na medida do necessário.

Um analisador Yacc [illegible] produz para cada token um valor inteiro associado. Se um token é declarado na primeira seção da especificação Yacc, o valor de atributo associado a um token é controlado através da variável definida pelo Yacc [illegible].

A partir do texto da página 456 e tokens de caracteres de um dígito cada, que usam a função [illegible]. Seus caracteres [illegible] e [illegible] e retornado. Devolva o texto [illegible] como o token [illegible].

Usando o Yacc com Gramáticas Ambíguas

Vamos agora explicar especificação Yacc de [illegible] que cada dígito dessa resultante [illegible]. Primeiro, [illegible] uma sequência de expressões numa [illegible] também permitir linhas em branco entre expressões. Devemos [illegible] a primeira regra para

[illegible]

Um Yacc [illegible] alternativa vazia, como [illegible].

Segundo, vamos expandir a classe de expressões de forma a incluir números em lugar de dígitos isolados e também os operadores aritméticos `-` (menos unário e binário), `*` e `/`. A primeira das 15 s/r de especificação desta classe de expressões é usar a gramática ambígua

[illegible] Número

Na especificação Yacc resultante é mostrada na Fig. 4.57.

Uma vez que a gramática da especificação da Fig. 4.57 é ambígua, o gerador LALR irá gerar conflitos de ações sintáticas. Yacc irá reportar o número de conflitos de ações sintáticas quando for criado o conjuntos de itens LR(0) e conflitos de ações sintáticas pode ser resolvido usando as regras de Yacc, como veremos a seguir. Isto significa que um argumento adicional [illegible] que [illegible] os conjuntos de itens encontrados na análise sintática LALR e uma representação legível da tabela sintática LR, mostrando como os conflitos entre ações sintáticas foram resolvidos. Sempre que Yacc relata que encontrou uma ação sintática [illegible] ele [illegible] para verificar [illegible] se [illegible] foram [illegible] se [illegible] corretamente.

A técnica que seguiremos daqui em diante [illegible] todos os conflitos de ações sintáticas usando duas regras simples:

1. Um conflito de redução/redução é resolvido através da escolha da produção com [illegible] Isto [illegible] a ordem na especificação Yacc. Por conseguinte, para se tomar a decisão correta na gramática de composição de [illegible], [illegible] [illegible] a produção [illegible] [illegible] da produção [illegible].
2. Um conflito deslocamento/redução é resolvido em favor do deslocamento. Esta [illegible] [illegible] conflito empilhará [illegible] que [illegible], [illegible] gramática [illegible] [illegible] [illegible].

Uma vez que essas regras *default* podem [illegible] sempre [illegible] o [illegible] do compilador de [illegible] Yacc providencia [illegible] mais [illegible] geral para resolver conflitos de empilhar/reduzir. Na parte de declarações, podemos associar precedências e associatividades aos terminais. A declaração

[illegible]

faz com que [illegible] [illegible] [illegible]

Fig. 4.57. Especificação Yacc para uma calculadora mais avançada

e podemos forçar um operador a ser binário não associativo existe. Estas diversas diretivas do operador nunca podem ser combinadas mais de uma vez.

Nas diretivas de precedência podemos colocar qualquer número de terminais, incluindo a forma literal. Os tokens na mesma declaração possuem a mesma precedência. Consequentemente, a declaração

```
%left '+' '-'
```

no Fig. 4.57 confere ao token '+' um nível de precedência igual ao daquele dos tokens nas declarações precedentes.

Yacc resolve os conflitos empilhar/reduzir comparando uma precedência associativa atribuída a cada produção envolvida num conflito empilhar/reduzir para cada terminal potencialmente envolvido. Se tiver que escolher entre empilhar um símbolo de entrada a e reduzir através da produção $A \to \alpha$, Yacc reduz se a precedência da produção for maior do que aquela de a ou se as precedências forem as mesmas e a associatividade da produção for %left. De outra forma, empilhar será a ação escolhida.

Normalmente, a precedência de uma produção é considerada a mesma que a do terminal mais à direita. Esta característica é satisfeita na maioria dos casos. Por exemplo, dadas as produções

$$E \to E - E \mid E * E \mid \ldots$$

poderíamos reduzir através de $E - E \cdot$ se o lookahead for '-', porque '-' ao lado direito possui a mesma precedência que o lookahead, mas gostaríamos de empilhar. Como lookahead '*' preferiríamos empilhar porque o lookahead possui maior precedência do que a produção.

Nas raras situações em que o terminal cuja precedência deveria ser a da produção não é adequado, uma precedência pode ser forçada anexando-se ao fim da produção uma precedência de uma pseudo-terminal:

```
expr : '-' expr %prec UMINUS
```

A precedência e associatividade da produção será então a mesma que a daquele terminal, que presumivelmente está definido nas seções de declarações. O Yacc não relata os conflitos de empilhar/reduzir que são resolvidos usando-se este mecanismo de precedência e associatividade.

Este terminal pode ser um qualificado de forma arbitrária. Na Fig. 4.57, este terminal nunca é retornado pelo analisador léxico mas é declarado somente para definir uma precedência para a produção. Na Fig. 4.57, a declaração

```
%right '^'
```

atribui uma precedência que é maior do que aquela de '*'. Na parte de regras de tradução, o símbolo

```
%prec UMINUS
```

ao final da produção

```
| '-' expr %prec UMINUS { $$ = -$2; }
```

faz com que o operador menos unário nesta produção tenha uma precedência maior do que a de qualquer outro operador.

Criando Analisadores Léxicos Yacc com Lex

Lex foi projetado para produzir analisadores léxicos que podem ser usados com Yacc. A biblioteca Lex [illegible] um programa diretor de análise [illegible] gerado pelo Yacc para ser uma rotina léxica. Se Lex for usado para produzir tokens de léxico, subordinados a uma [illegible] na terceira parte da especificação de Yacc, por comando

[illegible]

e lembrarmos que cada função Lex retorna um token conhecido pelo Yacc. Pelo mesmo comando [illegible] o programa [illegible] de definições inteiras do arquivo de saída Lex é compilado e como parte do arquivo de saída [illegible].

Sob o sistema UNIX, se a especificação Lex estiver no arquivo [illegible] e a especificação Yacc em [illegible] podemos obter

[illegible]

para obter o analisador desejado.

A especificação Lex na Fig. 4.58 pode ser usada em lugar do analisador léxico na Fig. 4.54. O último padrão [illegible] que Lex reconhece qualquer caractere que não seja um espaço em branco, tabulação ou fim de linha.

[illegible]

Fig. 4.58. Especificação Lex para [illegible] da Fig. 4.57.

Recuperação de Erros em Yacc

Em Yacc, a recuperação de erros pode ser realizada utilizando-se uma forma de regra de produção de erro. Primeiro, o usuário decide quais não-terminais "anais importantes" têm uma recuperação de erros associada a eles. As escolhas típicas são para algum tipo de não-terminais que gerem expressões, enunciados, blocos e procedimentos. Usuário escolhe adicionar a gramática produções de erro da forma $A \rightarrow$ error α, onde A é um não-terminal importante e α é uma cadeia de símbolos gramaticais, possivelmente uma cadeia vazia. error é uma palavra reservada. Yacc, Yacc vai gerar um analisador sintático a partir da especificação, tratando as produções de erro como produções comuns.

Durante um [illegible] analisador sintático gerado por Yacc encontra um erro, ele pode executar ações especificadas pelas produções de erro de uma forma especial. Ao encontrar um erro, Yacc desempilha símbolos até que encontre um estado s no topo da pilha para o qual exista um item da forma $A \rightarrow \cdot$ error α. O analisador sintático "empurra" um token fictício error como se o token estivesse sido encontrado.

Quando α é vazio, a ação para [illegible] imediatamente é a ação semântica associada à produção $A \rightarrow$ error. Esse tipo de produção é uma forma de recuperação de erros específica pelo usuário. O analisador sintático passa a desempilhar símbolos até que ele encontre um estado a partir do qual a análise sintática normal possa prosseguir.

Se α não for vazio, Yacc pula à frente na entrada procurando por uma subcadeia que possa ser reduzida a α. Se α for uma cadeia contendo apenas terminais, procura por essa cadeia de terminais na entrada e a reduz se encontrada. Nesse ponto, o analisador sintático trata error como se tivesse sido [illegible] e retoma a análise sintática normal.

Por exemplo, uma produção típica de error é, talvez,

$$stmt \rightarrow \text{error};$$

especificando que o analisador sintático deverá saltar até o próximo ponto-e-vírgula após encontrar um erro e assumir que um comando foi encontrado. A rotina semântica para esta produção de erro não necessitaria manipular a entrada, mas poderia gerar uma mensagem de diagnóstico e estabelecer um sinalizador para inibir a geração de código objeto, por exemplo.

Exemplo 4.59. A Fig. 4.59 mostra a calculadora de nova Yacc da Fig. 4.57 com a produção de erro

[illegible]

[illegible]

Fig. 4.59. Calculadora de nova com recuperação de erros.

Esta produção de erro faz com que a calculadora de mesa suspenda a análise sintática quando um erro for encontrado na entrada. Ao encontrar o erro, o analisador sintático da calculadora de mesa começa a remover símbolos da pilha até que encontre um estado que tenha uma ação de empilhar sobre o token **error**. O estado 0 é um desses estados (neste exemplo, é o único estado), uma vez que seus itens incluem

$$linhas \rightarrow \cdot\ \textbf{error}\ '\backslash n'$$

Igualmente, o estado 0 está sempre no fundo da pilha. O analisador sintático empilha o token **error** e então prossegue para saltar à frente na entrada até que tenha encontrado um caractere de avanço de linha. Nesse ponto o analisador sintático empilha o caractere de avanço de linha, reduz **error** '\n' para *linhas* e emite a mensagem de diagnóstico "redigite a última linha:". A rotina especial do Yacc yyerrok restabelece o analisador sintático em seu modo normal de operação. □

EXERCÍCIOS

4.1 Considere a gramática

$$S \rightarrow (\ L\ )\ |\ a$$
$$L \rightarrow L\ ,\ S\ |\ S$$

a) Quais são os terminais, não-terminais e o símbolo de partida?
b) Encontre as árvores gramaticais para as seguintes sentenças:
i) (a, a)
ii) (a, (a, a))
iii) (a, ((a, a), (a, a)))
c) Construa uma derivação mais à esquerda para cada uma das sentenças em (b).
d) Construa uma derivação mais à direita para cada uma das sentenças em (b).
e) Que linguagem esta gramática gera?

4.2 Considere a gramática

$$S \rightarrow aSbS\ |\ bSaS\ |\ \epsilon$$

a) Mostre que esta gramática é ambígua, construindo duas derivações mais à esquerda diferentes para a sentença *abab*.
b) Construa as derivações mais à direita correspondentes para *abab*.
c) Construa as árvores gramaticais correspondentes para *abab*.
d) Que linguagem esta gramática gera?

4.3 Considere a gramática

bexpr → *bexpr* **or** *bterm* | *bterm*
bterm → *bterm* **and** *bfactor* | *bfactor*
bfactor → **not** *bfactor* | (*bexpr*) | **true** | **false**

a) Construa uma árvore gramatical para a sentença **not** (**true** **or** **false**).
b) Mostre que esta gramática gera todas as expressões booleanas.
*c) Esta gramática é ambígua? Por quê?

4.4 Considere a gramática

$$R \rightarrow R\ '|'\ R\ |\ RR\ |\ R^*\ |\ (\ R\ )\ |\ a\ |\ b$$

Note que a primeira barra vertical é o símbolo "ou", e não um separador entre alternativas.
a) Mostre que esta gramática gera todas as expressões regulares sobre os símbolos *a* e *b*.
b) Mostre que essa gramática é ambígua.
*c) Construa uma gramática equivalente não-ambígua que confira aos operadores *, concatenação e | as precedências e associatividades definidas na Seção 3.3.
d) Construa uma árvore gramatical em ambas as gramáticas para a sentença *a*|*b***c*.

4.5 A seguinte gramática para os enunciados **if-then-else** é proposta para remediar a ambiguidade do else vazio:

cmd → **if** *expr* **then** *cmd*
| *cmd_associado*
cmd_associado → **if** *expr* **then** *cmd_associado* **else** *cmd*
| **outro**

Mostre que esta gramática ainda é ambígua.

***4.6** Tente projetar uma gramática para cada uma das seguintes linguagens. Quais delas são regulares?
a) O conjunto de todas as cadeias de 0's e 1's tais que cada 0 é imediatamente seguido por pelo menos um 1.
b) Cadeias de 0's e 1's com um número igual de 0's e 1's.
c) Cadeias de 0's e 1's com um número desigual de 0's e 1's.
d) Cadeias de 0's e 1's nas quais 011 não figura como uma subcadeia.
e) Cadeias de 0's e 1's da forma *xy* onde *x* ≠ *y*.
f) Cadeias de 0's e 1's da forma *xx*.

4.7 Construa uma gramática para as expressões de cada uma das seguintes linguagens:
a) Pascal
b) C
c) Fortran 77
d) Ada
e) Lisp

4.8 Construa gramáticas não-ambíguas para as sentenças de cada uma das linguagens do Exercício 4.7.

4.9 Podemos usar operadores semelhantes a expressões regulares nos lados direitos de produções gramaticais. Colchetes podem ser usados para denotar uma parte opcional de uma produção. Por exemplo, poderíamos escrever

cmd → **if** *expr* **then** *cmd* [**else** *cmd*]

para denotar um comando else opcional. Em geral, $A \rightarrow \alpha\ [\ \beta\ ]\ \gamma$ é equivalente às duas produções $A \rightarrow \alpha\beta\gamma$ e $A \rightarrow \alpha\gamma$.
Chaves podem ser usadas para denotar uma frase que possa ser repetida zero ou mais vezes. Por exemplo,

cmd → **begin** *cmd* { ; *cmd* } **end**

denota uma lista de comandos separados por ponto-e-vírgula e envolvidos por **begin** e **end**. Em geral, $A \rightarrow \alpha\ \{\ \beta\ \}\ \gamma$ é equivalente a $A \rightarrow \alpha B\gamma$ e $B \rightarrow \beta B\ |\ \epsilon$.
Num certo sentido, [β] figura no lugar da expressão regular β | ε e { β } no de β*. Podemos generalizar essas notações de forma a permitir quaisquer expressões regulares de símbolos gramaticais no lado direito de produções.
a) Modifique a produção *cmd* acima de tal forma que uma lista terminada com ponto-e-vírgula de *cmd*'s apareça no lado direito.
b) Forneça um conjunto de produções livres de contexto que gere o mesmo conjunto de cadeias que $A \rightarrow B^*a(C\ |\ D)$.
c) Mostre como substituir qualquer produção $A \rightarrow r$, onde *r* é uma expressão regular, por uma coleção finita de produções livres de contexto.

4.10 A seguinte gramática gera declarações para um único identificador:

cmd	→	**declare id** *lista_de_opções*
lista_de_opções	→	*lista_de_opções opção* | ε
opção	→	*modo* | *escala* | *precisão* | *base*
modo	→	**real** | **complex**
escala	→	**fixed** | **floating**
precisão	→	**single** | **double**
base	→	**binary** | **decimal**

a) Mostre como esta gramática pode ser generalizada de forma a permitir *n* opções A_i, $1 \le i \le n$, cada uma das quais podendo ser a_i ou b_i.

b) A gramática anterior permite declarações repetidas em comandos tais como

Poderíamos insistir em que a análise da linguagem proíba tais declarações. Somos, então, levados a um número infinito de sequências de letras que são sintaticamente corretas. Obviamente, essas declarações legais formam uma linguagem livre de contexto, de fato um conjunto regular. Escreva uma gramática para declarações com n opções, cada opção ligando no máximo uma vez.

*! c) Mostre que a gramática para a parte do possuir pelo menos 2^n símbolos.

d) O que você acha a respeito da viabilidade de se manter uma não redundância entre a nova verificação entre as opções nas declarações através da definição sintática da linguagem?

4.11 a) Elimine a recursão à esquerda da gramática do Exercício 4.1.

b) Construa um analisador sintático preditivo para a gramática resultante. Mostre o comportamento do analisador sintático nas sentenças do Exercício 4.1.1b).

4.12 Construa um analisador sintático de descendência recursiva com retrocesso para a gramática do Exercício 4.2. Você pode construir um analisador sintático preditivo para esta gramática?

4.13 A gramática

$$S \rightarrow aSa \mid aa$$

gera todas as cadeias de a's de comprimento par, exceto a cadeia vazia.

a) Construa um analisador sintático descendente recursivo com retrocesso para esta gramática que trate a alternativa $S \rightarrow aa$ antes de $S \rightarrow aSa$. Mostre que o procedimento para S irá ter sucesso para 2, 4, ou 8 a's, mas falha para 6 a's.

*! b) Que linguagem o seu analisador sintático reconhece?

4.14 Construa um analisador sintático preditivo para a gramática do Exercício 4.3.

4.15 Construa um analisador sintático preditivo para a parte da gramática ambígua para as expressões regulares do Exercício 4.4.

*! **4.16** Mostre que nenhuma gramática LL(1) à esquerda pode ser LL(1).

*! **4.17** Mostre que nenhuma gramática LL(1) pode ser ambígua.

4.18 Mostre que uma gramática sem produções ϵ na qual cada alternativa começa por um terminal distinto é sempre LL(1).

4.19 Um símbolo gramatical X é *inútil* se não existir derivação da forma $S \Rightarrow^* \alpha X \beta \Rightarrow^* \alpha\gamma\beta$, isto é, X não precisa ser aparecer na derivação de qualquer sentença.

a) Escreva um algoritmo para eliminar todas as produções contendo símbolos inúteis de uma gramática.

b) Aplique seu algoritmo à gramática

$$S \rightarrow 0 \mid A$$
$$A \rightarrow A1$$
$$B \rightarrow 1$$

4.20 Dizemos que uma gramática é *reduzida* se não possui produções ϵ nem existe exatamente uma produção $S \rightarrow \alpha$ e o símbolo inicial S não aparece em nenhum lado direito de qualquer produção.

a) Escreva um algoritmo para converter uma dada gramática numa gramática reduzida equivalente. *Sugestão*: Determine primeiro todos os não terminais que podem gerar cadeias de terminais.

b) Aplique seu algoritmo à gramática do Exercício 4.2.

4.21 Uma produção *unária* é aquela com um único não terminal do lado direito.

a) Escreva um algoritmo para converter uma gramática numa outra equivalente que não tenha produções unárias.

b) Aplique seu algoritmo à gramática do de expressões (4.10).

4.22 Uma gramática *livre de ciclos* não possui derivações da forma $A \Rightarrow^+ A$ para qualquer não terminal A.

a) Escreva um algoritmo para converter uma gramática numa outra livre de ciclos que seja livre de ciclos.

b) Aplique seu algoritmo à gramática $S \rightarrow SS \mid (S) \mid \epsilon$.

4.23 a) Quando a gramática do Exercício 4.1.1c) é construída para detectar comandos declarados repetidos, é possível decidir de cada forma sentencial derivada

b) Mostre os passos de um analisador sintático descendente recursivo correspondentes à derivação mais à direita de 4.

c) Mostre, num processo de compilação, onde ocorre a detecção de uma variável declarada repetidamente. Sugestão: Use uma tabela de símbolos.

4.24 A Fig. 4.60 mostra as relações de precedência de operadores para a gramática do Exercício 4.1. Quando essas relações de precedência analisam sintaticamente as sentenças do Exercício 4.1.1b)?

	a	(	)	,	$
a			$\cdot>$	$\cdot>$	$\cdot>$
(	$<\cdot$	$<\cdot$	$\doteq$	$<\cdot$	
)			$\cdot>$	$\cdot>$	$\cdot>$
,	$<\cdot$	$<\cdot$	$\cdot>$	$\cdot>$	
$	$<\cdot$	$<\cdot$			

Fig. 4.60 Relações de precedência para a gramática do Exercício 4.1.

4.25 Encontre as funções de precedência de operadores para a tabela da Fig. 4.60.

4.26 Existe uma forma mais sucinta de se produzir relações de precedência de operadores a partir de uma gramática de operadores incluindo aquelas com ambos lados direitos terminais. Defina $\mathit{primeiro}(A)$ para cada não terminal A, como sendo o conjunto de terminais a tal que a cadeia terminal $a\ldots$ esteja em alguma cadeia derivada a partir de A e defina $\mathit{ultimo}(A)$ como o conjunto de terminais que possam ser a última cadeia terminal à direita de alguma cadeia derivada a partir de A. Então, para terminais a e b, $a \doteq b$ sempre que $A \rightarrow \alpha a \beta b \gamma$ ocorra em P. Além disso, $a <\cdot b$ sempre que $A \rightarrow \alpha a B \beta$ ocorra em P e $b \in \mathit{primeiro}(B)$, ou $A \rightarrow \alpha a B b \beta$ ocorra em P, onde B é um não terminal, para todo $b \in \mathit{primeiro}(B)$. Finalmente, $a \cdot> b$ sempre que $A \rightarrow \alpha B b \beta$ ocorra em P e $a \in \mathit{ultimo}(B)$, onde B é um não terminal para todo $a \in \mathit{ultimo}(B)$.

a) Para a gramática do Exercício 4.1, compute $\mathit{primeiro}$ e ultimo para S e A.

b) Verifique que as relações de precedência da Fig. 4.60 são aquelas derivadas a partir desta regra.

4.27 Gere relações de precedência de operadores para as seguintes gramáticas.

a) A gramática do Exercício 4.2.

b) A gramática do Exercício 4.3.

c) A gramática de expressões (4.10).

4.28 Construa um analisador sintático de precedência de operadores para expressões regulares.

4.29 Uma gramática é denominada *gramática de operadores* se não existir produções ϵ nem existir produção com dois não terminais adjacentes do lado direito. Isto é, todo lado direito é uma cadeia de terminais ou de terminais separados por não terminais. Uma gramática de operadores pode ser ambígua. Por exemplo, a gramática $E \rightarrow E + E \mid E * E \mid (E) \mid a$ é uma gramática de operadores ambígua. Para cada gramática de operadores, podemos definir relações de precedência de operadores entre pares quaisquer de terminais. Quais das gramáticas do Exercício 4.27 são gramáticas de precedência de operadores?

4.30 Uma gramática é dita estar na *forma normal de Greibach* ou FNG se todas as produções são da forma $A \rightarrow a\alpha$, onde a é um terminal e α é uma cadeia de não-terminais (possivelmente vazia).

*c) Escreva um algoritmo para converter uma gramática na sua forma normal de Greibach equivalente.

b) Aplique seu algoritmo para converter a gramática de expressões do Exercício 4.1.

4.31 Mostre que toda gramática pode ser convertida numa gramática de operadores equivalente. *Sugestão*: transforme primeiro a gramática para a forma normal de Greibach.

4.32 Mostre que toda gramática pode ser convertida para uma gramática de operadores na qual cada produção está numa das seguintes formas:

$$A \to aBCA \qquad A \to aBC \qquad A \to aB \qquad A \to a$$

Se o vazio λ pertencer à linguagem gerada, $A \to \lambda$ também será uma produção.

4.33 Considere a gramática ambígua

$$S \to (S)\ S$$
$$S \to S\,[\,S\,]\ S$$
$$S \to \lambda$$

a) Construa a coleção de conjuntos LR(0) para esta gramática.

b) Construa um AFN no qual cada estado seja um item LR(0) de 4.1. Mostre que o grafo de dependência da coleção canônica de itens LR(0) para esta gramática é o mesmo que o AFD construído a partir do AFN usando a construção de subconjuntos.

c) Construa a tabela sintática usando o algoritmo SLR 4.x.

d) Mostre todos os movimentos permitidos pela tabela para a cadeia de entrada ([ ]).

e) Construa a tabela sintática canônica.

f) Construa a tabela sintática canônica usando o algoritmo LALR 4.13.

g) Construa a tabela sintática usando o Algoritmo LALR 4.14.

4.34 Construa uma tabela sintática SLR para a gramática do Exercício 4.3.

4.35 Considere a seguinte gramática

$$I \to I - E \mid T$$
$$E \to (I) \mid \hat{}\, E$$
$$T \to 0 \mid 1$$

a) Construa a tabela sintática SLR para esta gramática.

b) Construa a tabela sintática LALR.

4.36 Compare as tabelas sintáticas construídas nos Exercícios 4.33, 4.34 e 4.35, de acordo com os critérios da Seção 4.7.

4.37 a) Mostre que a seguinte gramática

$$S \to A\,a\,b\,c \mid B\,b\,B\,a$$
$$A \to \lambda$$
$$B \to \lambda$$

é LALR(1) mas não é SLR(1).

b) Mostre que esta gramática é LR(1) mas não é LALR(1).

*c) Mostre que esta gramática é LR(1) mas não é LR(0).

4.38 Mostre que nenhuma gramática LR(1) pode ser ambígua.

4.39 Mostre que a seguinte gramática

$$S \to A\,a \mid b\,A\,c \mid B\,c \mid b\,B\,a$$
$$A \to d$$
$$B \to d$$

é LALR(1) mas não é SLR(1).

4.40 Mostre que a seguinte gramática

$$S \to A \mid a\,b\,a \mid B\,a \mid b\,B\,c$$
$$A \to a$$
$$B \to a$$

é LR(1) mas não é LALR(1).

*4.41 Considere a família de gramáticas G_n definida por

$$S \to A_1 \mid B_1$$
$$A_i \to a\,A_{i+1} \mid a\,A_{i+1}\,b_i \qquad 1 \le i < n$$
$$A_n \to a \mid a\,b_n$$
$$B_i \to a\,B_{i+1}\,b_{i+1} \mid a\,B_{i+1}\,b_i \qquad 1 \le i < n$$
$$B_n \to a\,b_{n+1} \mid a\,b_n$$

a) Mostre que G_n possui $2n+4$ produções e $2^{n+1}+n-1$ conjuntos de itens LR(0). Que este resultado diz a respeito do tamanho que uma analisador sintático LR pode chegar a ter, comparado com o tamanho da gramática?

b) $G_n \in$ SLR(1)?

c) $G_n \in$ LALR(1)?

4.42 Escreva um algoritmo para computar, para cada não-terminal A numa gramática, o conjunto de não-terminais B tais que $A \Rightarrow^* \beta B$ para algumas cadeias β de símbolos gramaticais.

4.43 Escreva um algoritmo para computar, para cada não-terminal A numa gramática, o conjunto de terminais a tais que $A \Rightarrow^* \alpha a$ para algumas cadeias de terminais α, onde o último passo da derivação não use uma produção λ.

4.44 Construa um analisador sintático SLR para a gramática do Exercício 4.4. Resolva os conflitos de ações sintáticas de tal forma que as expressões regulares sejam analisadas sintaticamente da forma usual.

4.45 Construa um analisador sintático SLR para a gramática do Exercício 4.7, descrevendo o tratamento especial que você deu a conflitos de ações sintáticas da forma usual.

4.46 a) Construa uma tabela sintática SLR para a gramática

$$E \to E\ \textbf{sub}\ R\ J\ \textbf{sup}\ R \mid E \mid \mathrm{id}$$
$$R \to \mathrm{id} \mid \textbf{(}\ E\ \textbf{)}$$

Resolva os conflitos de ações sintáticas de tal forma que as expressões sejam agrupadas da mesma forma que pelo analisador sintático LR da Fig. 4.52.

b) Pode todo conflito de tipo redução/redução ser gerado no processo de construção da tabela sintática LR, ser convertido numa conflito do tipo empilha/reduz através da transformação da gramática?

*4.47 Construa uma gramática LR que valorize para a gramática de composição de tipos (4.25), que faz as expressões da forma E **sub** I **sup** E serem o caso especial.

4.48 Considere a seguinte gramática ambígua para os operadores binários infixos:

$$E \to E\ \mathrm{op}\ E \mid (\ E\ ) \mid \mathrm{id} \qquad \mathrm{op} \to + \mid - \mid * \mid / \mid \uparrow$$

Assuma que todos os operadores sejam associativos à esquerda e que $\uparrow$ tenha precedência sobre $*$ e $/$, e $*$ e $/$ sobre $+$ e $-$.

a) Construa o conjunto de itens SLR para esta gramática. Quantos conjuntos de itens existem, comparado com a função do item 4.1?

b) Construa a tabela sintática SLR para esta gramática e compare-a usando a representação de listas da Seção 4.7. Qual é o crescimento total de todas as listas (tabela mais representação), como função de n?

c) Quantos passos são consumidos para decompor $\mathrm{id} \uparrow \mathrm{id} \uparrow \mathrm{id}$?

*4.49 Repita o Exercício 4.48 para a gramática ambígua

$$E_0 \to E_0\ \mathrm{op}_0\ E_0 \mid E_1$$
$$E_1 \to E_1\ \mathrm{op}_1\ E_1 \mid E_2$$
$$\cdots$$
$$E_{n-1} \to E_{n-1}\ \mathrm{op}_{n-1}\ E_{n-1} \mid E_n$$
$$E_n \to (\ E_0\ ) \mid \mathrm{id}$$

O que as suas respostas aos Exercícios 4.48 e 4.49 dizem a respeito da eficiência relativa dos analisadores sintáticos para gramáticas ambíguas e não-ambíguas? O que sobre a eficiência relativa para construir os analisadores sintáticos?

4.50 Escreva um programa Yacc que reconheça expressões aritméticas como entrada e produza como saída as expressões pós-fixas correspondentes.

4.51 Escreva um programa de "calculadora de mesa" Yacc que vá avaliar expressões booleanas.

4.52 Escreva um programa Yacc que informe uma expressão regular como entrada e produza sua árvore gramatical como saída.

4.53 Resolva os mesmos problemas que serão feitos pelos analisadores sintáticos preditivos de precedência de operadores e LR dos Exemplos 4.52 e 4.53 para as seguintes entradas erradas:
a) `if id then if id then id else id`
b) `if id then id then id`

***4.54** Construa analisadores sintáticos de precedência de operadores e LR com correção de erros para a seguinte gramática:

```
stmt → if e then stmt
     | if e then stmt else stmt
     | while e do stmt
     | begin list end
     | s
list → list ; stmt
     | stmt
```

***4.55** A programação do Exercício 4.54 pode ser estendida LR através da substituição das produções para `list` por

```
list → list ; stmt { }
stmt → stmt { }
```

Construa analisadores sintáticos preditivos de correção de erros para a gramática resultante.

4.56 Mostre como os procedimentos de erros analisadores sintáticos dos Exercícios 4.54 e 4.55 para as entradas incorretas
a) `if e then s ; if e then s end`
b) `while e do begin s ; if e then s end`

4.57 Projete analisadores sintáticos LR de precedência de operadores e preditivos com recuperação de erros aproximadamente ótima para as gramáticas do Exercício 4.54 e 4.55, usando o ponto-e-vírgula e `end` como fontes de sincronização. Mostre os comportamentos de seus analisadores sintáticos para as entradas incorretas do Exercício 4.56.

4.58 Na Seção 4.3 foi proposto um método por graus para determinar o conjunto de cadeias que poderiam ser descartadas por um modo de recuperação de erro em analisadores sintáticos de precedência de operadores:
a) Projete um algoritmo para determinar se o conjunto de cadeias descartadas é finito ou infinito, usando as informações de precedência.
b) Forneça um algoritmo para determinar se o conjunto de cadeias é finito ou infinito, usando as informações de precedência.
c) Aplique seus algoritmos pertinentes da parte (b) à gramática do Exercício 4.54.

****4.59** Existem alternativas para os analisadores sintáticos com correção de erros das Figs. 4.38, 4.39 e 4.45 e que produzem correção de erros mais ajustada, provavelmente envolvendo mais processamento sobre a pilha por ocasião de um erro? Em que circunstâncias a pilha poderia ser um recurso crucial para a correção de erros seguinte?

4.60 Considere a gramática não ambígua para a forma normal de Chomsky do Exercício 4.44.
a) Forneça um algoritmo para converter uma gramática numa equivalente na forma normal de Chomsky.
b) Aplique seu algoritmo à gramática de expressões LISP.

4.61 Dada uma gramática na forma normal de Chomsky e uma cadeia de entrada $x = a_1 a_2 \cdots a_n$, escreva um algoritmo para determinar se x está em $L(G)$. *Sugestão*: Construa programação dinâmica, inserindo em uma tabela $T[i, j]$ o conjunto $\{A \mid A \Rightarrow^* a_i \cdots a_{i+j-1}\}$. Verifique se x está em $L(G)$ se e somente se $S \in T[1, n]$.

****4.62** a) Dada uma forma normal de Chomsky para uma gramática G, mostre como adicionar produções para inserção, remoção e mutação de um único erro a qualquer cadeia de tal forma que a gramática resultante gere todas as possíveis cadeias de *tokens*.
b) Modifique seu algoritmo de (a) para que ele corrija k erros de tal forma que dada qualquer cadeia x tal que $x \in L(G)$ ou x está a um erro de $L(G)$, seu algoritmo gera a lista de todas as cadeias em $L(G)$ que estão a k erros de x.

4.63 Escreva um analisador sintático Yacc para uma das gramáticas que usou no exercício de recuperação de erros do Exemplo 4.50.

NOTAS BIBLIOGRÁFICAS

O relatório Environment Agency (ANSI [1966]) introduz a forma normal de Backus-Naur (BNF) para definir a sintaxe de uma linguagem com muitas de preparação. A equivalência da BNF e das gramáticas livres de contexto foi rapidamente notada, e a teoria das linguagens formais recebeu uma grande parte de suas direções nos anos 60. Hopcroft e Ullman [1979] cobrem as bases desse campo.

Os métodos de análise sintática se tornaram muito mais sistemáticos após o surgimento das gramáticas livres de contexto. Várias técnicas gerais de análise sintática de qualquer gramática livre de contexto foram propostas. Uma das mais antigas é a da programação dinâmica sugerida por Cocke e Younger [1967] e Kasami [1965]. Outra tese de Ph.D. Early [1970] também descreveu uma algoritmo mais geral de análise sintática para todas as gramáticas livres de contexto. Aho e Ullman [1972a, 1973a] discutem esses e outros métodos de decomposição gramatical em detalhe.

Muitos diferentes métodos de análise gramatical têm sido empregados nos compiladores. Sheridan [1959] descreve o método de passo sintático usado no compilador Fortran original que adicionava parênteses adicionais em volta dos operandos de forma a tornar fácil de analisar expressões. A noção da precedência dos operadores e as funções de precedência de Floyd [1963]. Nos anos 60, um grande número de estratégias de análise sintática foram propostas, incluindo a precedência simples (Wirth e Weber [1966]), precedência mista (Floyd [1964], Graham [1964]), precedência fraca (McKeeman, Horning e Wortman [1970]) e precedência fraca (Ichbiah e Morse [1970]).

A análise sintática de descendência recursiva é a mais antiga técnica de análise sintática que foi usada em muitos sistemas de escrita de compiladores por muitos anos, incluindo MET (Schorre [1964]) e TMG (McClure [1965]). Uma solução para o exercício 4.17 pode ser encontrada em Birman e Ullman [1973]. Uma interessante gramática de análise sintática (Earley [1973]) projetada especialmente para a análise sintática de precedência de operadores.

As gramáticas LL foram estudadas por Lewis e Stearns [1968] e suas propriedades foram desenvolvidas em Rosenkrantz e Stearns [1970]. Algoritmos que fazem o uso eficiente dessas gramáticas foram descritos por Knuth [1971a], Lewis e Rosenkrantz e Stearns [1976] e

[illegible] dos analisadores sintáticos preditivos dos compiladores. Algoritmos para transformar as gramáticas para a forma LL(1) são apresentados em Foster [1968], Wood [1969], Stearns [1971] e Soisalon-Soininen e Ukkonen [1979].

As gramáticas e os analisadores correspondentes LR foram primeiramente estudados por Knuth [1965], que descreveu a construção de tabelas sintáticas LR canônicas. O método LR não foi considerado praticável por Korenjak [1969], mostrou que, com o mesmo, analisadores razoavelmente dimensionados poderiam ser produzidos para gramáticas de linguagens de programação. Quando DeRemer [1969, 1971] discutiu os métodos SLR e LALR, que são mais simples que o de Korenjak, a técnica LR se tornou o método de escolha para os geradores automáticos de analisadores sintáticos. Hoje em dia, geradores de analisadores sintáticos LR são componentes padrões de construção de compiladores.

Uma grande parte das pesquisas foi para engendrar os analisadores sintáticos LR. O uso de gramáticas ambíguas na análise sintática LR é devido a Aho, Johnson e Ullman [1975] e Earley [1975a]. A [illegible], o [illegible] tabelas sintáticas de produções simples foi estudado em Aho e Ullman [1973], Aho e Ullman [1973b], Demers [1974], Backhouse [1976], Johnson [1976], Pager [1977b], Soisalon-Soininen [1980] e Takeuchi [1981].

As técnicas para construção e ajustes de *footballs* LALR foram propostas por Lalonde [1971], Anderson, Eve e Horning [1973], Pager [1974] Soisalon-Soininen e Malmi [1981], DeRemer e Pennello [1982] e Park, Choe e Chang [1985], que também provaram algumas afirmações experimentais.

Aho e Johnson [1974] iniciaram uma pesquisa geral em análise sintática. Eles descreveram alguns dos algoritmos usados na geração de analisadores sintáticos. Você encontrará o uso de produções de erro para a recuperação de erros. Aho e Ullman [1972b] e [1973a] fornecem um excelente tratamento da análise sintática LR e de seus aspectos teóricos.

Muitas das técnicas de recuperação de erros para analisadores sintáticos foram propostas. As técnicas de recuperação de erros são pesquisadas por Cheatham [1979] e por Sippu [1981]. Irons [1963] propôs um esquema baseado na gramática para a recuperação de erros sintáticos. As produções de erro foram empregadas por Wirth [1968] para o tratamento de erros num compilador PL 360 [Wirth 1976] propôs a estratégia de recuperação em nível de frase. Aho e Peterson [1972] mostram como uma recuperação de erros de menor custo global pode ser alcançada utilizando-se produções de erro em conjunto com esquemas para gerar palavras livres de contexto. Mauney e Fischer [1982] estenderam esses trabalhos para a computação de menor custo local em analisadores sintáticos LL e LR, usando a técnica de análise sintática de Graham, Harrison e Ruzzo [1980]. Graham e Rhodes [1975] discutem a recuperação de erros no contexto da análise sintática de precedência de operadores.

Horning [1976] discute as qualidades que as mensagens de erro devem ter. Sippu e Soisalon-Soininen [1983, 1985] fornecem um desenvolvimento da técnica de recuperação de erros no *Finite Predictive Parsing* [Processadores de Linguagem Helsinki/Rauta et al. 1983] com a técnica de recuperação de "mínimo movimento para a frente" de Pennello e DeRemer [1978], a técnica de recuperação de erros de Graham, Haley e Joy [1979] e a técnica de recuperação de erros de "contexto global" de Pai e Kieburtz [1980].

A correção de erros durante a análise sintática é discutida por Conway e Maxwell [1963], Moulton e Muller [1967], Conway e Wilcox [1973], Levy [1975], Tai [1978] e Rubinch [1980]. Aho e Peterson [1972] fornecem uma solução para o Exercício 7.6.1

CAPÍTULO 5

TRADUÇÃO DIRIGIDA PELA SINTAXE

Este capítulo descreve a técnica da Seção 2.3, a tradução de linguagens guiada por gramáticas livres de contexto. Associamos informações a uma construção de linguagem de programação através de associar atributos aos símbolos gramaticais que representam a construção. Os valores para esses atributos são computados através de regras semânticas associadas às produções da gramática.

Existem duas notações para associar regras semânticas com produções, definições dirigidas pela sintaxe e esquemas de tradução. As definições dirigidas pela sintaxe são especificações de alto nível para as traduções. Escondem muitos detalhes de implementação e liberam o usuário de especificar explicitamente a ordem na qual as traduções ocorrem. Os esquemas de tradução oferecem alguma ordem na qual as regras semânticas são avaliadas e assim permitem que alguns detalhes de implementação sejam explicitados. Usamos definições dirigidas pela sintaxe para especificar a verificação semântica, particularmente na determinação de tipos, e no Capítulo 8 para gerar código intermediário.

Conceitualmente, com as definições dirigidas pela sintaxe, nós simplesmente aplicamos as regras semânticas para construir a árvore gramatical e, em seguida, a percorremos da forma necessária para avaliar as regras semânticas em cada nó. Veja a Fig. 5.1. A avaliação das regras semânticas pode gerar código, salvar definições numa tabela de símbolos, emitir mensagens de erro, realizar quaisquer outras atividades. A tradução de um fluxo de tokens é o resultado obtido através da avaliação das regras semânticas.

Uma implementação não tem que seguir literalmente o delineado na Fig. 5.1. Casos específicos de definições dirigidas pela sintaxe podem ser implementados numa única passagem através da avaliação das regras semânticas. Algumas vezes, é possível fazer isso sem explicitamente construir uma árvore gramatical ou um grafo que exibam as dependências entre os atributos. Uma vez que a implementação em uma passagem é muito importante, ela será discutida com detalhes neste capítulo. Uma subclasse importante, chamada de definições "L-atribuídas", abrange virtualmente todas as traduções que podem ser realizadas sem a construção explícita de uma árvore gramatical.

5.1 DEFINIÇÕES DIRIGIDAS PELA SINTAXE

Uma definição dirigida pela sintaxe é uma generalização de uma gramática livre de contexto na qual cada símbolo gramatical possui um conjunto associado de atributos, particionado em dois subconjuntos, chamados de atributos sintetizados e atributos herdados daquele símbolo gramatical. Suponha que x seja um nó na árvore gramatical construída para uma árvore gramatical, sendo um registro com campos para armazenar informações, e a um atributo associado com aquele nó. O valor para um atributo a em um nó x é escrito $x.a$. O valor de um atributo sintetizado em um nó é computado a partir dos valores dos atributos dos filhos daquele nó na árvore gramatical. O valor de um atributo herdado é computado a partir dos valores dos atributos dos irmãos e pais daquele nó.

As regras semânticas estabelecem dependências entre os atributos cujos valores são representados por um grafo. A partir do grafo de dependências obtemos uma ordem de avaliação para as regras semânticas. A avaliação das regras semânticas define os valores dos atributos nos nós da árvore gramatical para uma dada cadeia de entrada. Uma regra semântica também pode ter efeitos colaterais, como, por exemplo, imprimir um valor ou atualizar uma variável global. Nada impede uma implementação de precisar construir explicitamente uma árvore gramatical ou um grafo de dependências; eles têm apenas que produzir a mesma saída para cada cadeia de entrada.

Uma árvore gramatical mostrando os valores dos atributos em cada nó é chamada de árvore gramatical anotada. O processo de computar os valores dos atributos de cada nó é chamado de *anotação* ou *decoração* da árvore.

Forma das Definições Dirigidas pela Sintaxe

Numa definição dirigida pela sintaxe, cada produção gramatical $A \to \alpha$ tem associada a ela um conjunto de regras semânticas da forma $b = f(c_1, c_2, \ldots, c_k)$

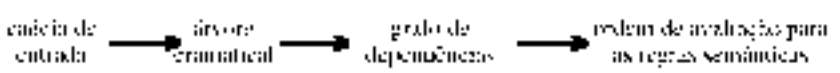


Fig. 5.1 Visão conceitual da tradução dirigida pela sintaxe

Produção	Regras Semânticas
$L \to E$ n	imprimir(E.val)
$E \to E_1 + T$	E.val = E_1.val + T.val
$E \to T$	E.val = T.val
$T \to T_1 * F$	T.val = T_1.val × F.val
$T \to F$	T.val = F.val
$F \to (E)$	F.val = E.val
$F \to$ dígito	F.val = dígito.lexval

Fig. 5.2. Definição dirigida pela sintaxe de uma calculadora de mesa simples

gramatical adotada para a produção $L \to E$ n. A saída imprime a raiz da árvore, E.val, que está no primeiro filho da raiz da árvore.

Para termos, fatores e dígitos, os atributos sintetizados são computados considerando primeiro o nó inferior mais à esquerda e mais ao fundo que corresponde ao lado direito da produção. $F \to$ **dígito**. A regra semântica correspondente F.val = **dígito**.lexval define o atributo F.val igual ao valor numérico inteiro do dígito. Por exemplo, se o dígito for **dígito**.lexval = 3, o filho direito de F na árvore à direita, o atributo F.val possui o valor 3 associado a esse F.

Considere a regra semântica para a produção $T \to T_1 * F$. O valor de atributo T.val é obtido pela multiplicação de

Produção	Regras Semânticas
$T \to T_1 * F$	T.val = T_1.val × F.val

Nos pontos em que a regra semântica é aplicada, T_1.val possui o valor 3, proveniente do filho à esquerda, e F.val o valor 5 do filho à direita. Por conseguinte, T.val adquire o valor 15 nesse nó.

A regra associada à produção para o não-terminal de partida $L \to E$ n imprime o valor da expressão gerada por E.

Atributos Herdados

Um atributo herdado é aquele cujo valor em um nó de uma árvore gramatical é definido em termos de atributos daquele nó, de seus irmãos e de seu pai. Os atributos herdados são úteis para expressar a dependência de uma construção de linguagem de programação no contexto em que ela aparece. Por exemplo, podemos usar um atributo herdado para controlar se um identificador aparece do lado esquerdo ou direito de uma atribuição, a fim de decidir se o endereço ou o valor de um identificador é necessário. Apesar de sempre possível fazer com que uma definição dirigida pela sintaxe contenha somente atributos sintetizados, esses definidos pela sintaxe com atributos herdados são frequentemente mais naturais.

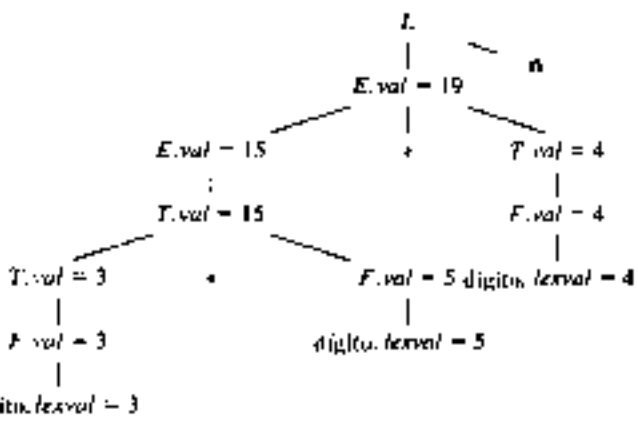


Fig. 5.3. Árvore gramatical anotada para

PRODUÇÃO	REGRAS SEMÂNTICAS
$D \rightarrow T\ L$	$L.in = T.type$
$T \rightarrow$ **int**	$T.type = integer$
$T \rightarrow$ **real**	$T.type = real$
$L \rightarrow L_1$, **id**	$L.in = L_1.in$
	acrescentar_tipo(**id**.*entrada*, $L.in$)
$L \rightarrow$ **id**	*acrescentar_tipo*(**id**.*entrada*, $L.in$)

Fig. 5.4. Definição dirigida pela sintaxe com $L.in$ como atributo herdado.

No exemplo a seguir, um atributo herdado distribui informações de tipo para os vários identificadores numa declaração.

Exemplo 5.4. Uma declaração gerada pelo não-terminal D numa definição dirigida pela sintaxe na Fig. 5.4 consiste na palavra-chave **int** ou **real**, seguida por uma lista de identificadores. O não-terminal T possui um atributo sintetizado $T.type$, cujo valor é determinado pela palavra-chave recebida. A regra semântica $L.in = T.type$, associada à produção $D \rightarrow T\ L$, faz o atributo herdado $L.in$ ganhar o tipo na declaração. As regras então propagam esse tipo pela árvore gramatical abaixo, usando o atributo herdado $L.in$. As regras associadas às produções para L chamam o procedimento *acrescentar_tipo* para incluir o tipo de cada identificador na sua entrada respectiva na tabela de símbolos apontada pelo atributo *entrada*.

A Fig. 5.5 mostra uma árvore gramatical anotada para a sentença **real** $\mathbf{id}_1, \mathbf{id}_2, \mathbf{id}_3$. Os valores de $L.in$ nos três nós L fornecem o tipo dos identificadores $\mathbf{id}_1$, $\mathbf{id}_2$ e $\mathbf{id}_3$. Esses valores são determinados pelo valor do atributo $T.type$ no filho à esquerda da raiz e são seguidos pela avaliação de $L.in$ de uma para cada um dos três nós da subárvore direita da raiz. A cada nó L, podemos também chamar o procedimento *acrescentar_tipo* para inserir na tabela de símbolos o tipo de cada identificador em cada filho à direita desse nó possuir o tipo real. —

Grafos de Dependências

Se um atributo b em um nó da árvore gramatical depender de um atributo c na regra semântica para b naquele nó precisa ser avaliada após a regra semântica que define c. As interdependências entre os atributos herdados e sintetizados nos nós da árvore gramatical podem ser determinadas através de um grafo chamado de *grafo de dependências*.

Antes de construir um grafo de dependências para uma árvore gramatical, colocamos cada regra semântica sob a forma $b = f(c_1, c_2, \ldots, c_k)$, através da introdução de um atributo sintético fictício b para cada regra semântica que consista em uma chamada de procedimento. O grafo possui um nó para cada atributo e um lado em direção ao nó para b a partir do nó para cada atributo c de que b depender. Mais detalhadamente, o grafo de dependências para uma dada árvore gramatical é construído como segue:

```
para cada nó na árvore gramatical faça
    para cada atributo desse nó gramatical, digamos a, faça
        acrescentar um nó no grafo de dependências para a
para cada nó na árvore gramatical faça
    para cada regra semântica b = f(c1, c2, ..., ck)
            associada à produção usada neste nó faça
        para i = 1 até k faça
            acrescentar um lado a partir do nó para ci até o nó para b
```

Por exemplo, suponha um nó A com $A.a = f(X.x, Y.y)$ e suponha que a regra semântica para a produção $A \rightarrow XY$ nesta regra defina o atributo sintetizado $A.a$ que depende dos atributos $X.x$ e $Y.y$. Se esta produção estiver sendo usada na árvore gramatical existirão no grafo de dependências três nós, $A.a$, $X.x$ e $Y.y$, com um lado para $A.a$ a partir de $X.x$ (uma vez que $A.a$ depende de $X.x$) e um lado para $A.a$ a partir de $Y.y$ (uma vez que $A.a$ também depende de $Y.y$).

Se a produção $A \rightarrow XY$ possuir a regra semântica $X.x = g(A.a, Y.y)$ associada a ela, existirá um lado para $X.x$ proveniente de $A.a$ e também uma lado para $X.x$ proveniente de $Y.y$, uma vez que $X.x$ depende de ambos $A.a$ e $Y.y$.

Exemplo 5.4. Sempre que a seguinte produção for usada numa árvore gramatical, adicionaremos os lados mostrados na Fig. 5.6 ao grafo de dependências.

PRODUÇÃO	REGRA SEMÂNTICA
$E \rightarrow E_1 * E_2$	$E.val = E_1.val \times E_2.val$

Os nós no grafo de dependências, marcados por $*$, representam os atributos sintetizados $E.val$, $E_1.val$ e $E_2.val$ nos nós correspondentes da árvore gramatical. O lado para $E.val$ a partir de $E_1.val$ mostra que $E.val$ também depende de $E_1.val$. As outras pontilhadas representam a árvore gramatical, mas não fazem parte do grafo de dependências. —

Exemplo 5.3. A Fig. 5.7 mostra o grafo de dependências para a árvore gramatical da Fig. 5.5. Os nós no grafo de dependências são marcados por números, esses números serão usados abaixo. Existe um lado em direção ao nó 5, associado a $L.in$ a partir do nó 4, associado a $T.type$, porque o atributo herdado $L.in$ depende do atributo $T.type$, de acordo com a regra semântica $L.in = T.type$ para a produção $D \rightarrow T\ L$. Os dois lados para baixo em direção aos nós 7 e 9 emergem porque $L.in$ depende de $L.in$ da segunda vez na regra semântica $L.in = L.in$ para a produção $L \rightarrow L_1$, **id**. Cada uma das regras semânticas *acrescentar_tipo*(**id**.*entrada*, $L.in$) associadas às produções L leva a criação de um lado fictício. Os nós fictícios não são construídos para esses atributos herdados. —

Ordem de Avaliação

Uma ordenação topológica de um grafo acíclico dirigido é qualquer ordenamento $m_1, m_2, \ldots, m_k$ dos nós do grafo, de tal forma que todos os lados partindo de um nó m_i para um nó m_j fazem com que m_i apareça antes de m_j na ordenação; isto é, $i < j$.

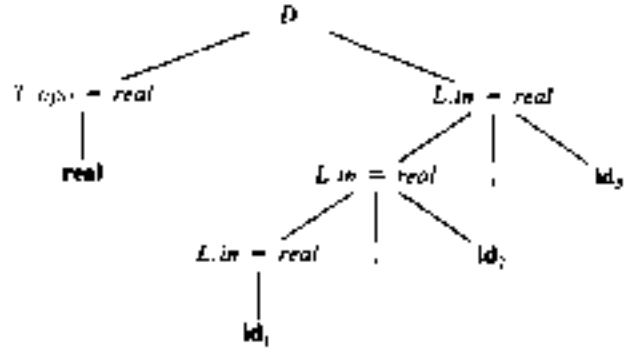


Fig. 5.5. Árvore gramatical com atributo herdado $L.in$ em cada nó rotulado L.

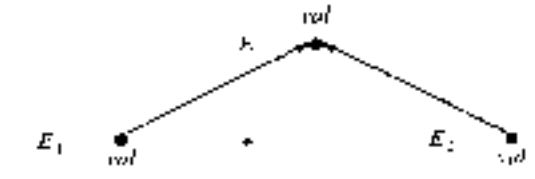


Fig. 5.6. Grafo de dependências para o nó $*$ em $E \rightarrow E_1 * E_2$.

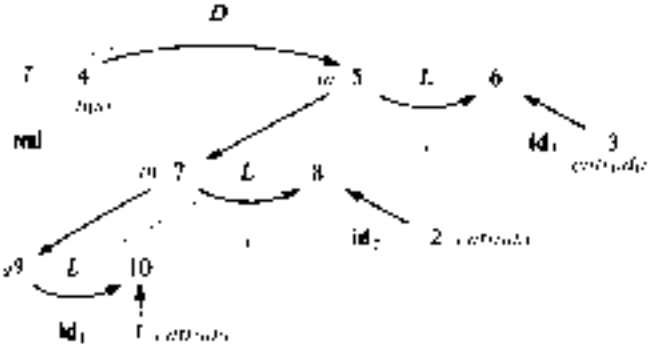


Fig. 5.7. Grafo de dependências para a árvore gramatical da Fig. 5.5

Qualquer ordenação topológica de um grafo de dependências fornece uma ordem válida na qual as regras semânticas associadas aos nós de uma árvore gramatical podem ser avaliadas. Ou seja, na classificação topológica os atributos $c.i$ e $c.s$, nos quais uma regra semântica $b := f(c.i, \ldots, c.s, \ldots)$ depende, estão disponíveis antes de f ser avaliada.

A tradução especificada por uma definição dirigida pela sintaxe pode ser obtida primeiro como segue. A gramática subjacente é usada para construir a árvore gramatical para a entrada. O grafo de dependências é construído como discutido acima. A partir de uma classificação topológica dos nós do grafo de dependências, obtemos a ordem em que as regras semânticas são avaliadas. A avaliação das regras semânticas nessa ordem produz a tradução da cadeia de entrada.

Exemplo 5.6. Cada um dos nós do grafo de dependências da Fig. 5.7 tem um número de numeração menor para um nó de numeração maior. Por exemplo, ordenações topológicas do grafo de dependências são obtidas escrevendo os nós pela ordem de seus números. A partir de uma classificação topológica obtemos o seguinte programa. Instruções para significar o atributo associado ao nó numerado n no grafo de dependências são

```
a1 := real
a2 := a1
coloque tipo em id3 entrada a2
a3 := a2
coloque tipo em id2 entrada a3
a4 := a3
coloque tipo em id1 entrada a4
```

A avaliação dessas regras semânticas armazena o tipo *real* na entrada da tabela de símbolos para cada identificador. □

Vários métodos foram propostos para a avaliação das regras semânticas:

1. *Algoritmos de árvores gramaticais.* Em tempo de compilação, esses métodos obtêm uma ordem de avaliação a partir da classificação topológica do grafo de dependências construído para a árvore gramatical para cada entrada. Esses métodos falham em encontrar uma ordem de avaliação somente se o grafo de dependências para a árvore gramatical particular contiver um ciclo.
2. *Métodos baseados em regras.* Em tempo de construção do compilador, as regras semânticas associadas às produções são analisadas manualmente ou por uma ferramenta especializada. Para cada produção é construída uma regra segundo a qual os atributos associados àquela produção são avaliados à medida em que o tempo de construção do compilador.
3. *Métodos oblivious.* Uma ordem de avaliação é escolhida sem levar em consideração as regras semânticas. Por exemplo, se a tradução ocorre durante a análise sintática, a ordem de avaliação é forçada pelo método de decomposição gramatical independentemente das regras semânticas. Uma avaliação oblivious restringe a classe de definições dirigidas pela sintaxe que podem ser implementadas.

Os métodos baseados em regras e oblivious não precisam construir explicitamente uma árvore gramatical e o tempo de compilação pode, então, ser mais eficiente no uso do tempo e do espaço durante a compilação.

Uma definição dirigida pela sintaxe é dita *circular* se o grafo de dependências para alguma árvore gramatical gerada por sua gramática contiver um ciclo. A Seção 5.10 discute como testar a circularidade de uma definição dirigida pela sintaxe.

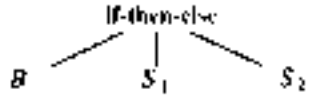
5.2 CONSTRUÇÃO DE ÁRVORES SINTÁTICAS

Nesta seção, mostramos como as definições dirigidas pela sintaxe podem ser usadas para especificar a construção de árvores sintáticas e outras representações gráficas das construções de linguagem.

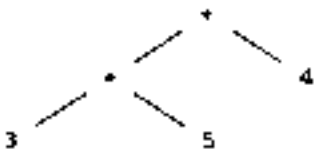
O uso de árvores sintáticas como uma forma de representação intermediária permite que a tradução seja desacoplada da análise sintática. As ordens de avaliação que são invocadas durante a análise sintática precisam conhecer exemplos dos tipos de restrições. Primeiro, uma gramática que seja adequada à análise sintática pode não refletir a estrutura hierárquica natural das construções da linguagem. Por exemplo, uma gramática para Fortran pode enxergar uma subcadeia como consistindo somente numa lista de números. Porém, a análise da subcadeia pode ser mais fácil se usarmos uma representação em árvore que reflita o atributo aplicado à lista. Segundo, uma ordem de análise sintática restringe a ordem na qual os nós na árvore gramatical são considerados. Essa ordem pode não ser adequada para a ordem na qual as informações sobre uma construção se tornam disponíveis. Por esta razão, os compiladores para C usualmente constroem árvores sintáticas para as declarações.

Árvores Sintáticas

Uma árvore sintática ilustrada é uma forma condensada de árvore gramatical, útil para representação das construções de linguagem. A produção $S \rightarrow$ **if** B **then** S_1 **else** S_2 poderia aparecer numa árvore sintática como



Numa árvore sintática, os operadores e palavras-chave não aparecem como folhas, mas em lugar são associados ao nó interior que seria o pai daquelas folhas na árvore gramatical. Outra simplificação encontrada nas árvores sintáticas é que cadeias de produções simples podem ser colapsadas. A árvore gramatical da Fig. 5.5 se torna árvore sintática



A tradução dirigida pela sintaxe pode ser usada em árvores sintáticas ou em árvores gramaticais. Identifique e comente em cada caso quais são os atributos aos quais nos referimos em árvore gramatical

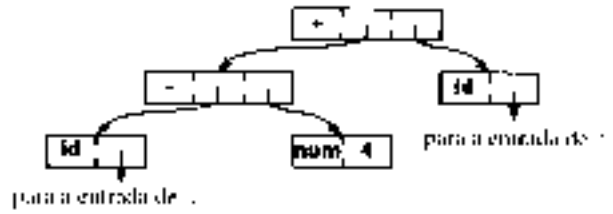


Fig. 5.8 Árvore sintática para a - 4 + c.

Construindo Árvores Sintáticas para Expressões

A construção de uma árvore sintática para uma expressão é similar à tradução de uma expressão para a forma posfixa. Construímos subárvores para as subexpressões através da criação de um nó para cada operador e operando. Os filhos de um nó operador são as raízes das duas subárvores que representam as subexpressões que constituem os operandos daquele operador.

Cada nó numa árvore sintática pode ser implementado como um registro com vários campos. No nó para um operador, um campo identifica o operador e os campos restantes contêm ponteiros para os nós dos operandos. O operador é frequentemente chamado o *rótulo* do nó. Quando usado numa tradução, os nós numa árvore sintática podem ter campos adicionais para guardar os valores (ou apontadores para os valores) dos atributos anexados ao nó. Neste seção, ilustraremos as seguintes funções para criar os nós das árvores sintáticas para as expressões com operadores binários. Cada função retorna um apontador para o nó recém-criado.

1. *criar_nó(op, esquerda, direita)* cria um nó de operador com rótulo *op* e dois campos contendo apontadores para *esquerda* e *direita*.
2. *criar_folha(id, entrada)* cria um nó de identificador com rótulo **id** e um campo contendo *entrada*, um apontador para a entrada da identificador na tabela de símbolos.
3. *criar_folha(***num***, val)* cria um nó número com rótulo **num** e um campo contendo *val*, o valor do número.

Exemplo 5.7. A seguinte sequência de chamadas de funções cria a árvore sintática para a expressão a - 4 + c na Fig. 5.8. Nesta sequência id_1, p_2, p_3, ... p_5 são apontadores para nós e *entrada_a* e *entrada_c* são apontadores para as entradas na tabela de símbolos dos identificadores a e c, respectivamente.

$p_1 = criar\_folha(\mathbf{id}, entrada\_a)$;
$p_2 = criar\_folha(\mathbf{num}, 4)$;
$p_3 = criar\_nó('-', p_1, p_2)$;
$p_4 = criar\_folha(\mathbf{id}, entrada\_c)$;
$p_5 = criar\_nó('+', p_3, p_4)$;

PRODUÇÃO	REGRAS SEMÂNTICAS
$E \rightarrow E_1 + T$	$E.nptr = criar\_nó('+', E_1.nptr, T.nptr)$
$E \rightarrow E_1 - T$	$E.nptr = criar\_nó('-', E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr = T.nptr$
$T \rightarrow (E)$	$T.nptr = E.nptr$
$T \rightarrow$ **id**	$T.nptr = criar\_folha(\mathbf{id}, id.entrada)$
$T \rightarrow$ **num**	$T.nptr = criar\_folha(\mathbf{num}, \mathbf{num}.val)$

Fig. 5.9 Definição dirigida pela sintaxe para construir árvores sintáticas para expressões.

A árvore é construída de baixo para cima. As chamadas de função *criar_folha(id, entrada_id)* e *criar_folha(***num***, val)* criam nós folha para **id**'s e **num**'s, respectivamente. Apontadores para esses nós são salvos com as regras associadas às produções $T \rightarrow$ **id** e $T \rightarrow$ **num**. Em seguida, a chamada *criar_nó('-', p1, p2)* constrói o nó interior com as folhas para a e 4 como filhos. Dois passos depois, *p3* está apontando para a raiz.

Uma Definição Dirigida pela Sintaxe para Construir Árvores Sintáticas

A Fig. 5.9 contém uma definição S-atribuída para construir uma árvore sintática por uma expressão contendo os operadores + e -. Todas as produções subjacentes da gramática são do tipo agrupado, e as chamadas para as funções *criar_nó* e *criar_folha* de modo a construir a árvore e os atributos sintetizados *nptr* contêm os apontadores retornados pelas chamadas de função.

Exemplo 5.8. Uma árvore gramatical anotada (uma representação de uma árvore sintática) para a expressão a - 4 + c é mostrada na Fig. 5.10. A árvore gramatical é exibida em linhas pontilhadas. Os nós da árvore gramatical rotulados pelos não-terminais E e T exporta atributo sintetizado *nptr*. A função de cópia que guarda o apontador para o nó da árvore sintática para uma expressão representada pelo não-terminal.

As regras semânticas associadas às produções $T \rightarrow$ **id** e $T \rightarrow$ **num** definem o atributo *T.nptr* como sendo um apontador para uma nova folha para o identificador ou um número respectivamente. Os atributos **id**.*entrada* e **num**.*val* são os valores lexicais que se espera que sejam retornados pelo analisador léxico junto com os tokens **id** e **num**.

Na Fig. 5.10, quando a expressão a - 4 + c é reduzida a E + T pela produção $E \rightarrow E + T$, o atributo *T.nptr* é a raiz da árvore para a - 4. Quando a regra semântica *E.nptr = criar_nó('+', E.nptr, T.nptr)* associada a essa produção é executada, *E.nptr* aponta para a raiz da árvore para a - 4 e *T.nptr* para a folha para c. Os dois apontadores passados para *criar_nó* são, portanto, transformados em filhos para o nó +.

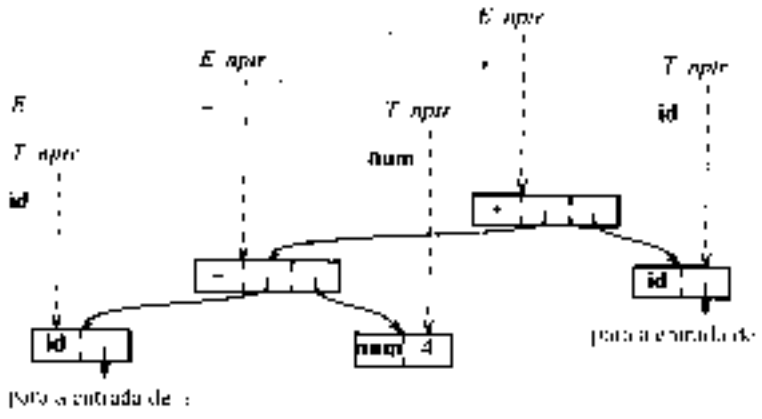


Fig. 5.10 Construção de uma árvore sintática para a - 4 + c.

Ao se interpretar a Fig. 5.10, é importante compreender que a *posição* de nível mais alto, ou ponto de ligação, é uma árvore sintática real que corresponde a cada esquema que a árvore produzida durante a análise gramatical, mas pode existir somente num sentido lógico. Na próxima seção, mostramos como uma definição Sequitur-tradicional pode ser implementada de forma simples usando-se a pilha de uma análise sintática bottom-up para controlar os valores dos atributos. Depois, com esta implementação, as ligações de construção de nós são executadas, da mesma maneira que na Fig. 5.7. ☐

Grafos Dirigidos Acíclicos

Um grafo dirigido acíclico criado por duplicação de nós de GDA para uma expressão identifica as subexpressões comuns existentes na mesma. Como uma árvore sintática, um GDA possui um nó para cada subexpressão de uma expressão; um nó interno representa um operador e os filhos representam seus operandos. A diferença está em que um nó em um GDA representando uma subexpressão comum possui mais de um "pai"; numa árvore sintática, a subexpressão comum seria representada por uma subárvore duplicada.

A Fig. 5.11 contém um GDA para a expressão

[illegible]

Nós do pai e pessoal dos pais porque [illegible] encontra duas subexpressões [illegible] igualmente [illegible] são representadas pelo mesmo nó, que tem dois pais.

A definição dirigida pela sintaxe da Fig. 5.9 na construção de GDA em lugar de uma árvore sintática só substitui todas as operações por construtores de nós. Um GDA é obtido se a função que constrói um novo nó primeiro procurar um nó que já tenha os argumentos dados. Por exemplo, antes de construir um nó com rótulo + e filhos p_1 e p_2, o construtor pode verificar se um nó com esses rótulos e filhos já foi construído. Se já foi construído, o construtor pode retornar um apontador para aquele nó previamente construído. A função de construção de folhas `criar_folha` pode ser feita de forma semelhante.

Exemplo 5.9. A sequência de operações na Fig. 5.12 constrói o GDA da Fig. 5.11 [illegible] `criar_nó` e `criar_folha` mantêm a rotina somente quando necessário, mantendo apontadores para os nós já existentes com rótulos e filhos corretos sempre que possível. Na Fig. 5.2, p_1, p_2, ..., p_{13} [illegible] rótulos na forma de símbolos para os dentes, [illegible] p_1 = [illegible]

Quando a chamada para `criar_nó(+, `**id**`, `**id**`)` é repetida a linha 2, o nó [illegible] pela chamada anterior `criar_nó(id, id)`, e retorna aquele nó, por exemplo, $p_2 = p_2$. Se a folha de nome, [illegible] retornados as folhas [illegible] são mesmos que aquelas retornadas pelas linhas 3 e 4, respectivamente. Consequentemente, a [illegible] na linha 10 precisa ser o mesmo que aquele construído pela chamada de `criar_nó` na linha 9. ☐

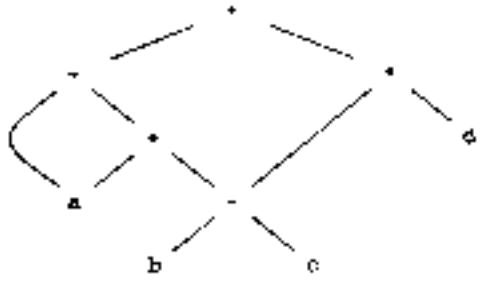


Fig. 5.11. GDA [illegible]

```
1)  p1 = criar_folha(id, a);
2)  p2 = criar_folha(id, a);
3)  p3 = criar_folha(id, b);
4)  p4 = criar_folha(id, c);
5)  p5 = criar_nó(*, p3, p4);
6)  p6 = criar_nó(*, p2, p5);
7)  p7 = criar_nó(+, p1, p6);
8)  p8 = criar_folha(id, b);
9)  p9 = criar_folha(id, c);
10) p10 = criar_nó(-, p8, p9);
11) p11 = criar_folha(id, d);
12) p12 = criar_nó(*, p10, p11);
13) p13 = criar_nó(+, p7, p12);
```

Fig. 5.12. Invocações para construir o GDA na Fig. 5.11

Em muitas aplicações, os nós são implementados como registros, [illegible] como na Fig. 5.13. Naquela figura, cada registro possui um campo de rótulo que determina a natureza do nó. Podemos nos referir a um nó por seu índice ou posição no array. O índice interno de um nó é frequentemente chamado de *número de valor*, por razões históricas. Por exemplo, usando os números de valor podemos dizer que o nó 3 possui o rótulo +, que seu filho à esquerda é o nó 1 e que seu filho à direita é o nó 2. O seguinte algoritmo pode ser usado a fim de criar os nós para uma representação sob a forma de GDA para uma expressão.

Algoritmo 5.1. Método dos números de valor para construir um nó numa GDA.

Suponhamos que os nós sejam armazenados num array, como na Fig. 5.13, e que cada nó seja referenciado através de seu número de valor. *Sempre encontramos* de um nó de operador uma tripla $\langle op, l, r \rangle$ consistindo em [illegible] e seu rótulo *op*, filho à esquerda *l* e filho à direita *r*.

Entrada. Rótulo *op* [illegible]

Saída. [illegible] com assinatura $\langle op, l, r \rangle$.

Método. Procurar no array por um nó com rótulo *op*, filho à esquerda *l* e filho à direita *r*. Se existir tal nó, retornar ele. Caso contrário, criar um novo nó com rótulo *op*, filho à esquerda *l* e filho à direita *r* e retornar *a*.

Uma forma óbvia de determinar se um nó [illegible] no array é manter todos os nós previamente criados numa lista e examinar cada nó para encontrá-lo pela assinatura desejada. A pesquisa pode ser tornada mais eficiente usando-se *k* listas, [illegible] de forma [illegible] usando uma função de *hash h* para determinar [illegible] a pesquisar.

A função de *hash h* computa o número do *bucket* a partir do valor de $\langle op, l, r \rangle$, [illegible] sempre retornar o mesmo número de *bucket*, dados os mesmos argumentos. Se o nó não estiver no *bucket* $h\langle op, l, r \rangle$, então nó novo nó é criado e adicionado ao mesmo, de forma que as pesquisas subsequentes o encontrarão lá. Vamos [illegible] os números podem colidir no mesmo número de *bucket*, mas na prática esperamos que cada *bucket* contenha somente um pequeno número de nós.

Cada *bucket* pode ser implementado como uma lista ligada como mostrado na Fig. 5.14. Cada célula numa lista ligada representa um nó. Os cabeçalhos de *buckets*, consistindo em apontadores para o primeira célula na lista, são armazenados num array. O número de

[illegible]

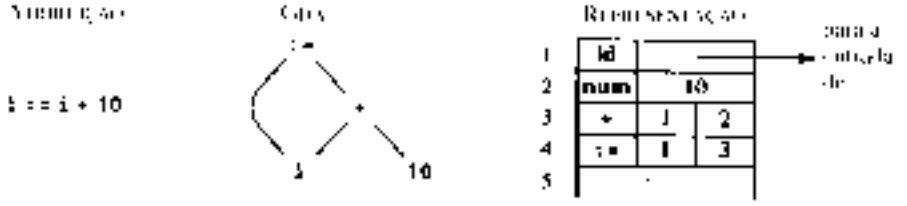


Fig. 5.13. Sequência de GDA para $x = i + 10$ ilustrando a construção de uma árvore sintática

nada é retornado pela função, o nó é indicado para essa versão de cabeçalhos de função.

Este algoritmo pode ser adaptado para ser aplicado à medida que nós são criados sequencialmente a partir de uma entrada. Em muitos compiladores, os nós são criados a medida que forem necessários, de forma a evitar a pré-alocação de um espaço que abrigue nós demais a maior parte do tempo e, em algum instante, insuficiente. Neste caso, não podemos assumir que os nós estejam localizados em memória sequencialmente e, por conseguinte, teremos que usar apontadores para nos referirmos aos mesmos. Se a função de *hash* puder ser estendida para computar o número do nó a partir de um rótulo e de apontadores para os filhos, podemos usar apontadores para os nós em lugar dos números de valor. Caso contrário, poderemos armazenar os nós de qualquer forma e usar esse número como o número de valor do nó.

Os GDAs podem também ser usados para representar programas ou expressões, uma vez que um GDA pode ter mais de uma raiz. Nos Capítulos 6 e 10 as computações realizadas por uma sequência de enunciados de atribuição serão representadas por um GDA.

5.3 AVALIAÇÃO *BOTTOM-UP* DE DEFINIÇÕES S-ATRIBUÍDAS

Assim que provarmos como usar as definições dirigidas pela sintaxe para especificar tradução, poderemos começar a pensar sobre como implementar os atributos para as mesmas. Pode ser útil desenvolvermos um algoritmo para uma definição dirigida pela sintaxe arbitrária. Infelizmente existem amplas classes de definições dirigidas pela sintaxe, mais pelas quais é fácil construir tradutores. Nesta seção, examinamos uma delas: as definições S-atribuídas, ou seja, as definições dirigidas pela sintaxe somente com atributos sintetizados. As seções subsequentes consideram a implementação de definições que contêm atributos herdados.

Os atributos sintetizados podem ser avaliados por um analisador sintático *bottom-up* na medida em que a entrada seja decomposta gramaticalmente. O analisador sintático computa os valores dos atributos sintetizados associados aos símbolos gramaticais em sua pilha. Sempre que uma redução for realizada, os valores dos novos atributos sintetizados são computados a partir dos atributos que figuram na pilha para os símbolos gramaticais na lado direito da produção reduzida. Esta seção constrói sobre a pilha do analisador sintático pode ser ampliada de forma a abrigar os valores desses atributos sintetizados. Veremos na Seção 5.6 que esta implementação também suporta alguns atributos herdados.

Somente os atributos sintetizados ligados a uma definição dirigida pela sintaxe da Fig. 5.9 para a construção de árvores sintáticas de uma expressão. A abordagem desta seção pode subsequentemente ser aplicada na construção de árvores sintáticas durante a análise sintática *bottom-up* descrita na Seção 5.2. A tradução de expressões durante a análise sintática *top-down* frequentemente usa atributos herdados. Por conseguinte, postergaremos a tradução durante a análise sintática *top-down* até que as dependências "esquerda-direita" separadamente na próxima seção.

Atributos Sintetizados na Pilha do Analisador Sintático

Um tradutor para uma definição S-atribuída pode frequentemente ser implementado como auxílio de um gerador de analisadores sintáticos LR, tal como aquele discutido na Seção 4.9. A partir de uma definição S-atribuída, o gerador de analisadores sintáticos pode construir um tradutor que avalie os atributos à medida que for reconhecendo gramaticalmente a entrada.

Um analisador sintático *bottom-up* utiliza uma pilha para armazenar informações sobre as subárvores que tenham sido estruturadas. Podemos usar campos extra na pilha do analisador sintático para abrigar os valores dos atributos sintetizados. A Fig. 5.15 mostra um exemplo de uma pilha sintática com espaço para um valor de atributo. Vamos supor, como na figura, que a pilha seja implementada através de um par de vetores *estado* e *valor*. Cada entrada de *estado* é um apontador (ou índice) de uma tabela sintática LR(1). Observe que o símbolo gramatical está implícito no estado e não precisa ser armazenado na pilha. Eventualmente, colocaremos referências ao estado através do mesmo símbolo gramatical que o mesmo sobre quando colocado na pilha sintática, de acordo com o descrito na Seção 5.2. Se o estado i no topo da pilha for A, então $val[i]$ irá abrigar o valor do atributo associado ao nó da árvore sintática correspondente a esse A.

O topo corrente da pilha é indicado através de apontador *topo*. Assumimos que os atributos sintetizados serão avaliados exatamente antes de cada redução. Suponhamos que a regra semântica $A := f(X, Y, Z)$ esteja associada à produção $A := XYZ$. Antes que XYZ seja reduzido a A, o valor do atributo Z está em $val[topo]$, o de Y em $val[topo - 1]$ e o de X em $val[topo - 2]$. Se um símbolo não possui atributo, então a entrada correspondente no vetor *val* é indefinida. Após a redução, *topo* é decrementado de 2, e o estado que cobre A é colocado

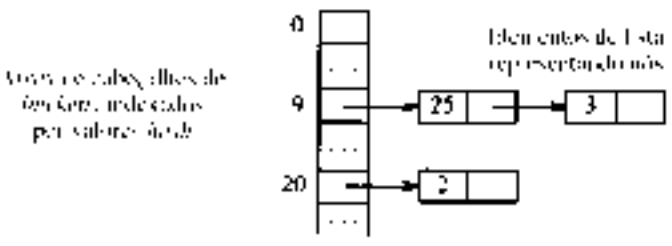


Fig. 5.14. Estrutura de dados para pesquisa em tabela de símbolos.

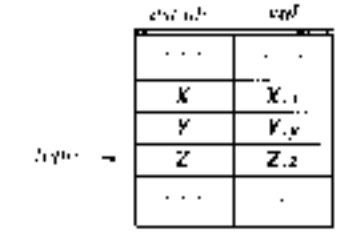


Fig. 5.15. Pilha de analisador sintático com espaço para atributos sintetizados.

ENTRADA	estado	val	PRODUÇÃO SEMÂNTICA
	–	–	
n	n	3	
n	F	3	$F \rightarrow \textbf{digito}$
n	T	3	$T \rightarrow F$
* n	T *	3 –	
digito n	T * digito	3 – 5	$F \rightarrow \textbf{digito}$
+ n	T	15	$E \rightarrow T$
+ n	E	15	$E \rightarrow E + T$
(n	(	15	$F \rightarrow (E)$
n	(n	15 –	
n	(F	15 – 4	
n	(T	15 – 1	$T \rightarrow F$
n	(T *	15 – 1 –	
n	(T * digito	15 – 1 – 2	$F \rightarrow \textbf{digito}$
)	(E)	15	$T \rightarrow (E)$
$	E	15	$S \rightarrow E$

Fig. 5.17. Movimentos efetuados pela tradução sintática

5.4 DEFINIÇÕES L-ATRIBUÍDAS

Quando a tradução ocorre durante a análise sintática, a ordem de avaliação dos atributos é ligada à ordem na qual os nós da árvore gramatical são "criados" pelo método de análise gramatical. Como mencionamos anteriormente, dois métodos de tradução *top-down* e *bottom-up*, é aquela rotulada através da aplicação do procedimento *reduzir* (Fig. 5.18) à raiz da árvore gramatical. Chamamos esta ordem de avaliação de *ordem de pesquisa em profundidade*. Ainda que a árvore gramatical nunca seja construída de fato, é útil estudar a tradução durante a análise sintática como se ela estivesse sendo construída. Nesta seção, vamos considerar uma classe de definições dirigidas pela sintaxe, chamada de definições L-atribuídas, cujos atributos podem ser avaliados em uma ordem de pesquisa em profundidade (o "L" está no lugar de "left", porque as informações de atributos aparecem em um arquivo de esquerda para a direita). A implementação de classes progressivamente mais amplas de definições L-atribuídas é coberta nas próximas três seções deste capítulo. As definições L-atribuídas incluem todas as definições dirigidas pela sintaxe baseadas nas gramáticas LL e LR. Na Seção 5.5 temos um modelo de implementação de definições numa única passagem usando métodos de análise sintática preditiva. Uma classe mais ampla de definições L-atribuídas é implementada na Seção 5.6 durante a análise sintática *bottom-up*. Ampliaremos os métodos da Seção 5.3. Um método geral de implementação de todas as definições L-atribuídas é descrito na Seção 5.7.

Definições L-Atribuídas

Uma definição dirigida pela sintaxe é *L-atribuída* se cada atributo herdado de X_j, $1 \leq j \leq n$, no lado direito de $A \rightarrow X_1 X_2 \cdots X_n$, depender somente

1. dos atributos dos símbolos $X_1, X_2, \ldots, X_{j-1}$ à esquerda de X_j na produção e
2. dos atributos herdados de A.

Note-se que cada definição S-atribuída é L-atribuída porque as restrições 1 e 2 se aplicam somente aos atributos herdados.

Exemplo 5.11. A definição dirigida pela sintaxe da Fig. 5.19 não é L-atribuída, porque o atributo herdado $Q.i$ do símbolo gramatical Q do

AÇÃO SEMÂNTICA	PRODUÇÃO
$print(val[top])$	$S \rightarrow E\ \$$
$val[ntop] := val[top] + val[top-2]$	$E \rightarrow E + T$
$val[ntop] := val[top] * val[top-2]$	$T \rightarrow T * F$
$val[ntop] := val[top-1]$	$F \rightarrow (E)$
$val[ntop] := val[top] - 1$	$F \rightarrow \textbf{digito}$

Fig. 5.16. Implementação de uma calculadora de mesa com um analisador sintático LR.

```
procedimento visitar(n : nó);
início
    para cada filho m de n, da esquerda para a direita faça
        início
            avaliar os atributos herdados de m;
            visitar(m);
        fim;
    avaliar os atributos sintetizados de n
fim
```

Fig. 5.18. Ordem de avaliação em profundidade para árvores de análise sintática.

Produção	Regras Semânticas
$A \rightarrow XY$	$X.i := f_1(A.i)$
	$Y.i := f_2(A.i, X.s)$
	$A.s := f_3(X.s, Y.s)$
$A \rightarrow BC$	$B.i := f_4(A.i)$
	$C.i := f_5(A.i, B.s)$
	$A.s := f_6(B.s, C.s)$

Fig. 5.19. Funções exigidas pelas regras semânticas.

pende do atributo S do antecedente, o qual é sintetizado. Outros exemplos de dependências que não são tão diretas podem ser encontrados nas Seções 5.6 e 5.9.

Esquemas de Tradução

Vimos pela primeira vez a tradução dirigida pela sintaxe no contexto em que os atributos são associados aos símbolos gramaticais e as ações semânticas são avaliadas sempre que uma produção é usada durante a análise sintática. Nesta seção, introduzimos a notação formal que pode ser usada para especificar a tradução dirigida pela sintaxe. Essa notação, chamada de esquema de tradução, é importante porque muitas vezes ela é a entrada para uma ferramenta que gera automaticamente um tradutor. Além disso, esquemas de tradução podem ser implementados durante a análise sintática sem a construção explícita de uma árvore de análise sintática.

Um esquema de tradução é uma gramática livre de contexto na qual os atributos são associados aos símbolos gramaticais e as ações semânticas são colocadas entre chaves { } e inseridas nos lados direitos das produções, como na Seção 2.3. Usamos os esquemas de tradução neste capítulo como uma notação útil para a especificação da tradução dirigida pela sintaxe.

Os esquemas de tradução considerados nesse capítulo também podem ter atributos sintetizados quando localizados. Nos esquemas simples da tradução orientada pela sintaxe do Capítulo 2, os atributos eram do tipo cadeia para cada símbolo e para cada produção $A \rightarrow X_1 X_2 \ldots X_n$ a ação semântica era uma cadeia formada pela concatenação das cadeias para $X_1, X_2, \ldots, X_n$, nessa ordem, com algumas cadeias adicionais opcionais entre elas. Vamos ver por que podemos realizar a tradução de modo mais simples quando todos os atributos são sintetizados. Mais tarde, estudaremos as regras mais gerais para esquemas de tradução.

Exemplo 5.12. Aqui está um esquema simples de tradução que converte expressões infixas, envolvendo adição e subtração, em expressões pós-fixas correspondentes. Esta é uma pequena revisão do trabalho feito no esquema de tradução (2.14) do Capítulo 2.

```
L → E fim
E → E + T { imprimir('+') }
E → E - T { imprimir('-') }
E → T
T → 0 { imprimir('0') }
T → 1 { imprimir('1') }
...
T → 9 { imprimir('9') }
```

A Fig. 5.20 mostra a árvore gramatical para a entrada 9 - 5 + 2 com ações semânticas anexadas apropriadamente como filhos de um nó que corresponde ao lado esquerdo da produção que aquela ação está associada. Dito de outra forma, estamos usando a mesma técnica de antes para construir uma árvore de análise sintática, mas estamos também incluindo ações como filhos de nós da árvore. Essas ações são executadas quando as arestas daquele nó para seus filhos são encontradas durante uma travessia em profundidade da árvore. Por exemplo, a ação imprimir('-') é executada entre as chamadas de procedimento visitar(9) e visitar(5), enquanto imprimir('+') é executada após visitar(5).

Quando desejamos implementar esquemas de tradução, precisamos determinar algumas restrições de forma a assegurar que um valor de atributo esteja disponível quando uma ação o referencia. Essas restrições, motivadas pelas dependências de atributos discutidas na Seção 5.1, garantem que ações possam ser executadas durante uma travessia em profundidade da árvore de análise sintática.

Uma das mais simples ocorre quando somente atributos sintetizados são associados. Para esse caso, podemos executar uma ação de tradução anexando-a à produção que consista numa atribuição para cada atributo sintetizado do símbolo do lado esquerdo. Cada regra calcula um atributo sintetizado para o não-terminal do lado esquerdo da produção, ou um atributo sintetizado para um símbolo gramatical do lado direito. Em qualquer caso, podemos implementar a ação colocando-a no fim do lado direito da produção.

Nas próximas duas seções, mostramos como um esquema de tradução que satisfaz essas restrições pode ser implementado durante a análise sintática. Nas Seções 5.4 e 5.5, estudamos esquemas de tradução mais gerais.

Quando todos os atributos são sintetizados, as ações de um esquema de tradução podem ser executadas durante uma travessia em profundidade da árvore de análise sintática. Em particular, as ações podem ser executadas durante uma análise sintática ascendente sempre que uma redução for realizada. Para ilustrar a tradução durante a análise sintática, considere o esquema de tradução do Exemplo 5.12. A execução das ações resulta na tradução 9 5 - 2 + para a entrada 9 - 5 + 2.

Para mais detalhes sobre esquemas de tradução, veja o Capítulo 2.

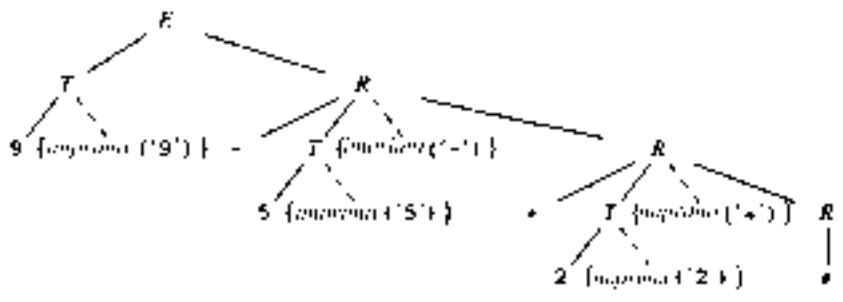


Fig. 5.20. Árvore gramatical para 9 - 5 + 2 com ações semânticas.

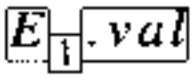


Fig. 5.21. Coleção de expressões aritméticas pela gramática $E \to E + T \mid T$, $T \to T * F \mid F$, $F \to (E) \mid \mathbf{digit}$.

O seguinte esquema de tradução não satisfaz a primeira dessas duas restrições:

$$A \to Q \; R \quad \{ R.i = f(Q.s) \}$$
$$A \to a \quad \{ A.s = g(a.v) \}$$

Presumimos que o atributo herdado $R.i$ na segunda produção não está definido explicitamente. Portanto, as regras de cópia podem ser usadas para dar a $R.i$ o valor $Q.s$, antes que os valores de $Q.s$ e $a.v$ sejam calculados. Se $a.v$ for uma representação de cadeias, como em um exemplo anterior, então $f(Q.s)$ poderia depender de cada caractere individual de $a.v$. Se A for avaliado durante a análise sintática *bottom-up*, então $g(a.v)$ poderia ser calculada durante a análise sintática, mas $f(Q.s)$ só poderia ser calculada durante a redução pela segunda produção.

É sempre possível começar com um esquema de tradução dirigida pela sintaxe e construir um esquema de tradução que satisfaça as duas condições. Esta seção começa com um exemplo ilustrando essa construção. Em seguida, apresenta um algoritmo de formatação de tabelas LaTeX, descrito inicialmente na Seção 1.3. Dada a entrada

```
.TS
C | N .
Nome | Número
Adams | 2
Jones | 13
.TE
```

LaTeX coloca C e N nas posições relativas e centraliza os números, como mostrado na Fig. 5.21. Note-se que o símbolo T é um processador de tabelas e bordas tabulares é uma ferramenta para baixo em relação a .TS e .TE.

Exemplo 5.13. A partir da definição dirigida pela sintaxe na Fig. 5.22, iremos construir um esquema de tradução da Fig. 5.23. Nas figuras, o não terminal B representa uma caixa gerada em LaTeX, uma abstração semelhante à usada em PIC, descrita no Exemplo 1.3. Dada a entrada

```
.TS
C | N .
Nome | Número
Adams | 2
Jones | 13
.TE
```

LaTeX coloca C e N nas posições relativas e centraliza os números, como mostrado na Fig. 5.21. Note-se que o símbolo T é um processador de tabelas e bordas tabulares é uma ferramenta para baixo em relação a .TS e .TE.

Exemplo 5.13. A partir da definição dirigida pela sintaxe na Fig. 5.22, iremos construir um esquema de tradução da Fig. 5.23. Nas figuras, o não terminal B representa uma caixa gerada em LaTeX, uma abstração semelhante à usada em PIC, descrita no Exemplo 1.3. A produção $B \to B\,B$ representa a justaposição de dois quadros. $B \to B$ **sub** B representa a colocação do segundo quadro sobre o primeiro, alinhando a base do segundo quadro na posição de subscrito do primeiro quadro. A produção $B \to$ **texto** representa um quadro cujo tamanho é determinado pela largura e altura de uma cadeia de texto. O atributo t de **texto** é obtido através de uma pesquisa em tabela, dando o caractere representado pelo token **texto**. Quando a produção $B \to B\;B_1$ é aplicada, B e B_1 herdam o tamanho do ponto-pai, usando as regras de cópia. A largura de B, representada pelo atributo sintetizado lg, é o maior das larguras de B e B_1.

PRODUÇÃO	REGRAS SEMÂNTICAS
$S \to B$	$B.ps = 10$
	$S.ht = B.ht$
$B \to B_1\,B_2$	$B_1.ps = B.ps$
	$B_2.ps = B.ps$
	$B.ht = \max(B_1.ht, B_2.ht)$
	$B.lg = B_1.lg + B_2.lg$
$B \to B_1$ **sub** B_2	$B_1.ps = B.ps$
	$B_2.ps = \mathit{shrink}(B.ps)$
	$B.ht = \mathit{disp}(B_1.ht, B_2.ht)$
	$B.lg = B_1.lg + B_2.lg$
$B \to$ **texto**	$B.ht = \mathbf{texto}.t \times B.ps$
	$B.lg = \mathbf{texto}.l \times B.ps$

Fig. 5.22. Definição dirigida pela sintaxe para os tamanhos e localização de caixas.

$S \to$		$\{ B.ps = 10 \}$
	B	$\{ S.ht = B.ht \}$
$B \to$		$\{ B_1.ps = B.ps \}$
	B_1	$\{ B_2.ps = B.ps \}$
	B_2	$\{ B.ht = \max(B_1.ht, B_2.ht);$
		$B.lg = B_1.lg + B_2.lg \}$
$B \to$		$\{ B_1.ps = \mathit{shrink}(B.ps) \}$
	B_1	
	sub	$\{ B_2.ps = \mathit{compose}(B.ps) \}$
	B_2	$\{ B.ht = \mathit{disp}(B_1.ht, B_2.ht);$
$B \to$ **texto**		$B.lg = \mathbf{texto}.l \times B.ps \}$

Fig. 5.23. Esquema de tradução correspondente à parte (b) da Fig. 5.22.

Quando a produção $B \to B_1$ **sub** B_2 é usada, a função *shrink* reduz o tamanho de ponto de B em 30%. A função *disp* deve permitir um deslocamento para baixo de cada um de B_1 e B_2. As regras que geram os comandos gráficos de campos são de tipos complexos, não são mostradas.

A definição da Fig. 5.22 é L-atribuída. O único atributo herdado é ps, do não terminal B. Cada regra semântica define ps como uma constante ou em termos do atributo herdado do nó-pai imediato à esquerda da produção. Por conseguinte, a definição é L-atribuída.

O esquema de tradução da Fig. 5.23, obtido através da inserção de atribuições correspondentes, as regras semânticas na Fig. 5.22 aparecem nas produções, segundo as três convenções anteriores. Para melhor ilustrar o algoritmo de tradução, cada não-terminal após a primeira produção aparece em sua própria linha, e as ações são alinhadas à direita. Consequentemente,

$$S \to \{ B.ps = 10 \}\; B \quad \{ S.ht = B.ht \}$$

é escrito assim:

$$S \to \{ B.ps = 10 \}$$
$$B \quad \{ S.ht = B.ht \}$$

Note-se que as regras que estabelecem os valores dos atributos herdados $B_1.ps$ e $B_2.ps$ ligam estritamente os nós de B e B_1 nos lados esquerdos das produções.

5.5 TRADUÇÃO *TOP-DOWN*

Nesta seção, as definições L-atribuídas serão implementadas durante uma análise gramatical preditiva. Trabalharemos com esquemas de tradução adequados de definições dirigidas pela sintaxe, de forma a termos ações explícitas a respeito da ordem na qual as ações e as alterações de atributos ocorrem. Estenderemos igualmente o algoritmo de eliminação da recursão à esquerda para os esquemas de tradução com atributos sintetizados.

Eliminação da Recursão à Esquerda de um Esquema de Tradução

Uma vez que o número dos operadores aritméticos é irrelevante para esquerda e direita, usaremos gramáticas recursivas à esquerda para expressões. Estenderemos o algoritmo para eliminação da recursão à esquerda nas Seções 2.4 e 4.3, de forma a permitir atributos quando a gramática subjacente de um esquema de tradução for transformada. A transformação se aplica a esquemas de tradução com atributos sintetizados. Permite que muitas das definições dirigidas pela sintaxe das Seções 2.4 e 5.2 sejam implementadas utilizando a análise sintática preditiva. O próximo exemplo ilustra a transformação.

Exemplo 5.14. O esquema de tradução da Fig. 5.24 é transformado através do esquema de tradução da Fig. 5.25. O novo esquema produz a árvore gramatical anotada da Fig. 5.26 para a expressão 9-5+2. As

$$
\begin{array}{lll}
L \rightarrow & E_1 + T & \{L.val = E_1.val + T.val\} \\
E \rightarrow & E_1 - T & \{E.val = E_1.val - T.val\} \\
E \rightarrow & T & \{E.val = T.val\} \\
T \rightarrow & (E) & \{T.val = E.val\} \\
 & \textbf{num} & \{T.val = \textbf{num}.val\}
\end{array}
$$

Fig. 5.24. Esquema de tradução com uma gramática recursiva à esquerda

setas na figura sugerem uma forma para a determinação desses valores da expressão.

Na Fig. 5.26(a), os números nodais são gerados pela árvore sintática. Cada nó tem um atributo chamado *val*. Cada folha tem o valor léxico fornecido pelo atributo **num**.*val*. O nó da subexpressão $9-5$ é gerado pelo E mais à esquerda, mas o seu valor, 4, é calculado após o nó gerado pelo R do filho à direita da raiz. O atributo herdado $R.i$ obtém o valor 9 a partir de $T.val$. A subtração $9-5$ é calculada quando o resultado 4 do nó do meio, para R, são feitas inserindo-se a seguinte ação entre T e R, em $R \rightarrow -\ T\ R$:

$$\{R_1.i = R.i - T.val\}$$

Uma ação similar adiciona 2 ao valor de $+\ T$, produzindo o resultado $R.i = 6$ para o nó de topo para R. Esse resultado é necessário à raiz como o valor de $E.val$. O atributo sintetizado s para R, não mostrado na Fig. 5.26, é usado para copiar o resultado apropriado até a raiz.

Para análise sintática *top-down*, pode-se assumir que uma ação seja executada no tempo em que um símbolo na mesma posição é expandido. Por conseguinte, na segunda produção da Fig. 5.23, a primeira ação (atribuição a $R.i$) é realizada após T ter sido completamente expandido em terminais e a segunda após R ter sido completamente expandido. Como explicado na discussão das definições L-atribuídas da Seção 5.4, um atributo herdado de um símbolo precisa ser computado por uma ação que ligue antes do símbolo e um atributo sintetizado de um não terminal à esquerda precisa ser computado após o cômputo de todos os atributos dos quais depende.

Como facilidade de aquisição, a análise sintática precisa a outros esquemas de tradução recursivos à esquerda, expressando-os em termos dos atributos $R.i$ e $R.s$ na Fig. 5.25, mais abstratamente. Suponhamos ter o seguinte esquema de tradução

$$
\begin{array}{lll}
A \rightarrow A_1\ Y & \{A.a = g(A_1.a, Y.y)\} \\
A \rightarrow X & \{A.a = f(X.x)\}
\end{array}
\qquad (5.21)
$$

Cada símbolo gramatical possui um atributo sintetizado escrito usando-se letra minúscula correspondente, e cada função é designada arbitrariamente. A generalização para produções A adicionais e para produções com cadeias em lugar dos símbolos X e Y pode ser feita como no Exemplo 5.15 adiante.

$$
\begin{array}{lll}
L \rightarrow & T & \{R.i = T.val\} \\
 & S & \{L.val = R.s\} \\
R \rightarrow & - & \\
 & T & \{R_1.i = R.i - T.val\} \\
 & R_1 & \{R.s = R_1.s\} \\
R \rightarrow & + & \\
 & T & \{R_1.i = R.i + T.val\} \\
 & R & \{R.s = R_1.s\} \\
R \rightarrow & \varepsilon & \{R.s = R.i\} \\
T \rightarrow & (& \\
 & E & \\
 &) & \{T.val = E.val\} \\
T \rightarrow & \textbf{digit} & \{T.val = \textbf{digit}.val\}
\end{array}
$$

Fig. 5.25. Esquema de tradução com ações para gramática adequada à análise sintática preditiva

O algoritmo para eliminar a recursão à esquerda na Seção 2.4 constrói a seguinte gramática a partir de (5.21):

$$
\begin{array}{l}
A \rightarrow X\ R \\
R \rightarrow Y\ R \mid \varepsilon
\end{array}
\qquad (5.22)
$$

Levando em conta as ações semânticas, o esquema transformado se torna

$$
\begin{array}{lll}
A \rightarrow X & \{R.i = f(X.x)\} & \\
 & R & \{A.a = R.s\} \\
R \rightarrow Y & R_1 & \{R.i = g(R.i, Y.y)\} \\
 & R & \{R.s = R_1.s\} \\
R \rightarrow \varepsilon & \{R.s = R.i\} &
\end{array}
\qquad (5.23)
$$

O esquema transformado usa os atributos i e s para R, como na Fig. 5.25. Para se ver por que os resultados de (5.21) e (5.23) são os mesmos, consideremos as árvores gramaticais anotadas na Fig. 5.27. O valor de $A.a$ é computado de acordo com (5.21) na Fig. 5.27(a) e na Fig. 5.27(b) contém os cômputos dos valores de $R.i$ e $R.s$ de acordo com (5.23). O valor de $R.i$ no filho à esquerda da raiz é $f(X.x)$, o valor herdado de acordo com (5.23). O valor de $R.i$ no filho à direita da raiz é $g(f(X.x), Y.y)$, o valor sintetizado de acordo com (5.23). Esse valor é mostrado na Fig. 5.27(b).

Exemplo 5.15. Se a definição dirigida pela sintaxe na Fig. 5.6 para construção de árvores sintáticas for convertida para um esquema de tradução, as produções e ações semânticas para L se tornam

$$
\begin{array}{lll}
L \rightarrow E\ \$ & \{L.nptr = \textbf{mknode}(\textbf{op}, \textbf{'\$'}, E.nptr, \textbf{nil})\} \\
E \rightarrow E_1 + T & \{E.nptr = \textbf{mknode}(\textbf{op}, \textbf{'+'}, E_1.nptr, T.nptr)\} \\
E \rightarrow T & \{E.nptr = T.nptr\}
\end{array}
$$

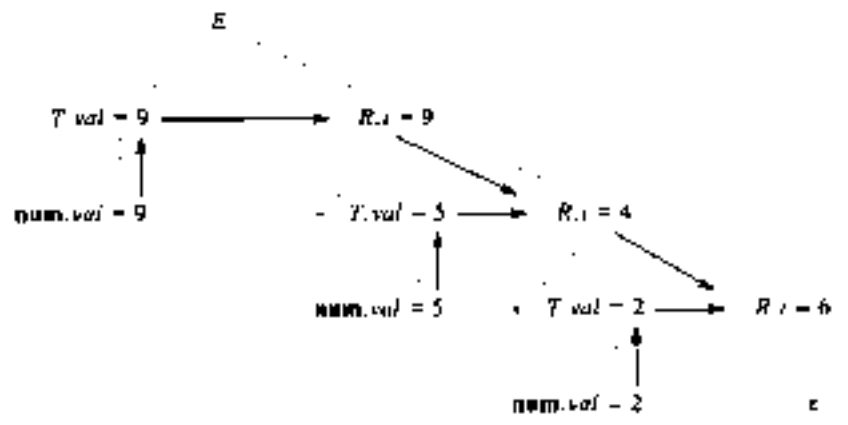


Fig. 5.26. Valores gerados e usados

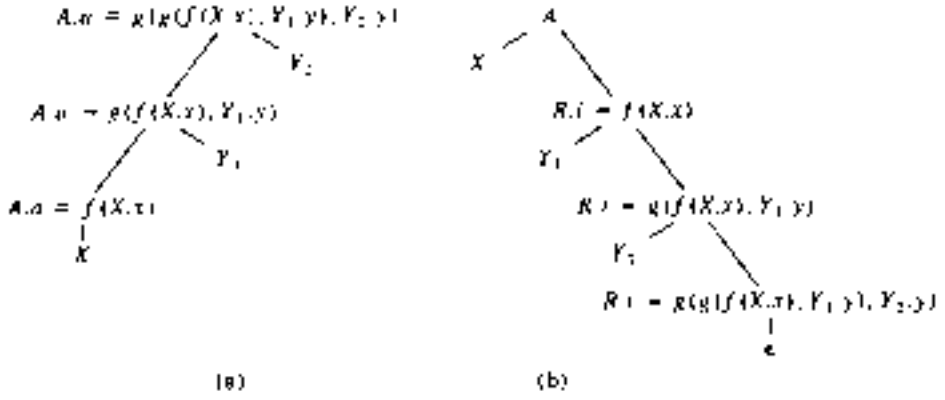


Fig. 5.27 (a) Árvores de computação para cadeias de atributos

Quando a expressão é esperada, o tradutor gerado segue o esquema de tradução. A variável *E* corresponde a *X* em 15.2) e, após a leitura de + (ou −), as duas primeiras produções correspondem a *T* como termo inicial *T* na terceira produção corresponde a *X*. O esquema de tradução transformado é o mostrado na Fig. 5.28. As produções e as ações semânticas para *E* assemelham-se àquelas na definição original, na Fig. 2.8.

A Fig. 5.29 mostra como as ações da Fig. 5.28 constroem a árvore sintática para 3 + 5. Os atributos sintetizados são mostrados à direita do ponto no símbolo gramatical e os atributos herdados são mostrados à esquerda. Uma vez que a expressão é uma lista de números, todas as folhas são produções $T \rightarrow$ **num**, como no Exemplo 5.5. Ao *T* mais à esquerda é atribuído *E.esqui* apontando para a folha de 3. Um operador *pai* é criado e atribuído como atributo *E.s* das folhas direita e esquerda de *E* → *T R*.

Quando a produção $R \rightarrow +\ T\ R_1$ é aplicada, o filho esquerdo da raiz *R* aponta para o nó criado *E.esqui* para o nó de 3. O nó para + é construído aplicando-se a função *criaNo* aos operadores e aos ponteiros correspondentes.

Finalmente, quando a produção $R \rightarrow \varepsilon$ é aplicada, *R* aponta para a raiz de toda a árvore sintática. Toda a árvore sintática é retornada através dos atributos *s* dos nós para *R*, não mostrados na Fig. 5.29, até que se torne o valor de *E.esqui*.

Projeto de um Tradutor Preditivo

O processo algorítmico para a construção de analisadores sintáticos preditivos, a partir de um esquema de tradução baseado numa gramática adequada à análise sintática preditiva

Algoritmo 5.2. Construção de um tradutor preditivo dirigido pela sintaxe

```
E → T {R.i = T.esqui}
    R {E.esqui = R.s}
R → + T {R1.i = criaNo('+', R.i, T.esqui)}
    R1 {R.s = R1.s}
  | - T {R1.i = criaNo('-', R.i, T.esqui)}
    R1 {R.s = R1.s}
  | ε {R.s = R.i}
T → (
    E
    )
  | num {T.esqui = criaFolha(num, num.val)}
```

Fig. 5.28 Esquema de tradução transformado para avaliação de árvores sintáticas

preditiva. (1) esquema de tradução dirigido pela sintaxe com uma gramática subjacente adequada à análise sintática preditiva

Saída. Código para um tradutor dirigido pela sintaxe.

Método. As ações para a modificação da construção de um analisador sintático preditivo da Seção 2.4:

1) Para cada não-terminal *A*, construa uma função que tenha um parâmetro formal para cada atributo herdado de *A*. A função retorna uma estrutura de registro contendo um campo para cada atributo sintetizado de *A*, passado pelo gerador como um campo para cada atributo, utilizando um mecanismo de passagem por referência, para a transmissão de parâmetros discutida na Seção 5.6. Por simplicidade, assumimos que cada não-terminal tenha exatamente um atributo sintetizado. A função para *A* possui uma variável local para cada atributo de cada símbolo gramatical que figure numa produção para *A*.

2) Como na Seção 2.4, o código para o não-terminal *A* decide qual produção usar, baseado no símbolo corrente de entrada.

3) O código associado a cada produção realiza o seguinte. Considere os símbolos, os ações e os atributos herdados do lado direito da produção, da esquerda para a direita.

   i) Para o símbolo *X* com atributo herdado *i*, salve o valor de uma variável local para *X.i*. Em seguida, gere uma chamada para reconhecer o token *X* e, se *X* for um não-terminal

   ii) Para o não-terminal *B*, gere uma atribuição $c = B(a_1, a_2, \ldots, a_k)$, com uma chamada de função no lado direito onde $a_1, a_2, \ldots, a_k$ são os valores para os atributos herdados de *B* e *c* é a variável para o atributo sintetizado de *B*.

   iii) Para uma ação, copie o código dentro do analisador sintático, substituindo cada referência a um atributo pelo valor para aquele atributo.

O Algoritmo 5.2 é estudado na Seção 5.7 de forma a implementar qualquer definição L-atribuída, uma vez que uma árvore sintática tenha sido construída. Na Seção 5.8, consideramos formas de reconhecimento ascendente construídos pelo Algoritmo 5.2. Por exemplo, torna-se possível eliminar marcadores de empilhamento e variáveis numa implementação pilha-valor para alguns esquemas de tradução. Algumas das implementações podem ainda ser feitas automaticamente pelos geradores de analisadores sintáticos descritos no Capítulo 10.

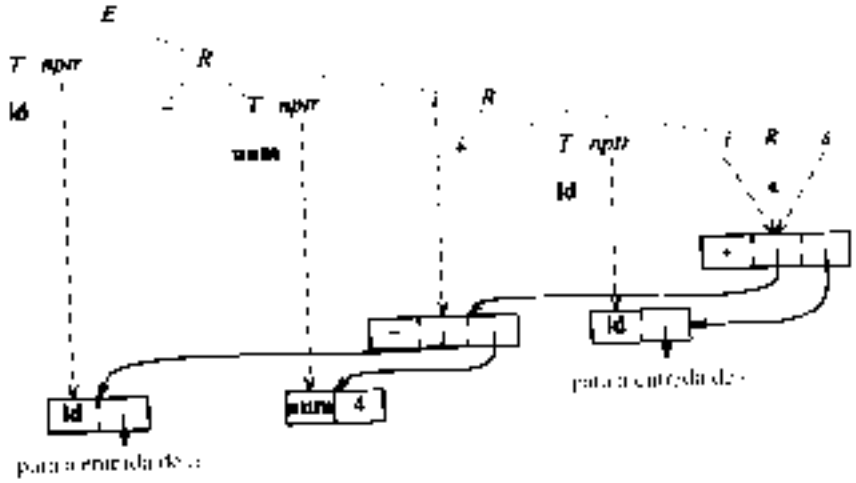


Fig. 5.29. Uso dos atributos herdados para construir árvores sintáticas.

Exemplo 5.16. A gramática da Fig. 5.28 é LL(1) e, por conseguinte, adequada à análise sintática preditiva. A partir dos atributos dos não-terminais da gramática, obtemos os seguintes tipos de argumentos e resultados para as funções de E, R e T. Lembre-se que E e T possuem dois atributos herdados, que passam a ser argumentos

```
função E: { nó_arvore_sintatica,
função R: { nó_arvore_sintatica } nó_arvore_sintatica
função T: { nó_arvore_sintatica }
```

Os subprogramas dessas três funções são para as produções R na Fig. 5.28 para testar o valor de **op_aditivo**. As novas produções usam o token **op_aditivo** para representar $+$ ou $-$:

$$
\begin{aligned}
R &\rightarrow \textbf{op\_aditivo}\ T\ \{R_1.h := \textbf{criar\_no}(\textbf{op\_aditivo}.lexema, R.h, T.nptr)\}\ R_1 \\
R &\rightarrow \varepsilon \quad \{R.s := R.h\} \\
\varepsilon &\rightarrow \varepsilon \quad \{R.s := R.h\}
\end{aligned}
\tag{5.5}
$$

A função para R está baseada no procedimento de análise sintática da Fig. 5.20. Se o símbolo de *lookahead* for **op_aditivo**, a produção $R \rightarrow$ **op_aditivo** $T\ R$ é aplicada utilizando-se o procedimento *reconhecer* e a variável *Lexema* para obter o valor de entrada após **op_aditivo** e, então, chamando os procedimentos para R e T. Em caso contrário, o procedimento não faz nada, pois isto dá a produção $R \rightarrow \varepsilon$.

O procedimento para R na Fig. 5.30 contém código para a avaliação de atributos. O valor léxico (*lexval*) do *token* **op_aditivo** é guardado em *lexema_op_aditivo*. **op_aditivo** é reconhecido, T é chamado e o resultado é guardado usando-se *ajuste*. A variável R corresponde ao atributo herdado $R_1.h$ e o atributo sintetizado $R_1.s$. O procedimento retorna o valor de s após alcançar o marcador de ação. As funções para E e T são construídas similarmente.

```
procedimento R;
início
    se lookahead = op_aditivo então início
        reconhecer(op_aditivo); T; R
    fim
    senão início {produzir nada}
    fim
fim;
```

Fig. 5.30. Procedimento de análise sintática para as produções $R \rightarrow$ op-aditivo $T\ R \mid \varepsilon$.

```
Função R(h: ↑nó_arvore_sintatica; ant: ↑nó_arvore_sintatica):
        ↑nó_arvore_sintatica;
    var op: (+, -); T: ↑nó_arvore_sintatica;
        lexema_op_aditivo: char;
início
    se lookahead = op_aditivo então início
        { produção R → op_aditivo T R }
        lexema_op_aditivo := lexval;
        reconhecer(op_aditivo);
        op := +;
        T := { chamada para o procedimento T };
        ajuste := R(h);
        s := ajuste
    fim
    senão s := h; { produção R → ε }
    retornar s
fim;
```

Fig. 5.31. Construção de nós em uma função baseada em procedimento.

5.6 AVALIAÇÃO *BOTTOM-UP* DOS ATRIBUTOS HERDADOS

Nesta seção, apresentamos um método de implementação de definições L-atribuídas sobre uma estrutura de análise sintática ascendente (*bottom-up*). O método é capaz de tratar todas as definições L-atribuídas consideradas até aqui, possivelmente, ao mudar o esquema que possa ser necessário. A definição L-atribuída baseada numa gramática LL(1) pode tipicamente ser implementada sob uma análise não-preditiva L-atribuída baseada numa gramática LR(1). O método é uma generalização da técnica de tradução *bottom-up* introduzida na Seção 5.5.

Removendo Ações Infiltradas dos Esquemas de Tradução

No método de tradução descrito na seção 5.5, consideramos esquemas de tradução nos quais as ações estavam associadas exclusivamente aos finais das produções, enquanto que no método de análise preditiva recursiva descendente as ações foram infiltradas em vários pontos ao longo do lado direito. Para construirmos tradutores sobre esquemas de tradução *bottom-up* para definições L-atribuídas, primeiro mostramos como transformar esquemas de tradução de modo que todas as ações ocorram nos finais das produções. Em seguida, introduzimos uma transformação que faz com que todas as ações ocorram nos finais das produções.

[illegible] ações utilizadas no esquema de tradução-exemplo com os lugares [illegible] de suas produções.

A [illegible] especificação [illegible] gerador [illegible] da gramática básica. Substituímos cada ação [illegible] por um [illegible] e [illegible] de [illegible]. [illegible] exemplo, o esquema de tradução

$L \rightarrow I \quad R$

$R \rightarrow$, I {*imprime*(*C*.*val*)} R_1 | ε

$I \rightarrow$ **num** {*imprime*(**num**.*val*)}

é transformado, usando-se dois terminais marcadores M e N, em

$$L \rightarrow I \quad R$$
$$R \rightarrow , \quad I \quad M \quad R_1 \qquad I \quad N \quad R_1 \mid \varepsilon$$
$$I \rightarrow \textbf{num} \ \{imprime(\textbf{num}.val)\}$$
$$M \rightarrow \varepsilon \ \{imprime(','\,)\}$$
$$N \rightarrow \varepsilon \ \{imprime('+'\,)\}$$

A gramática nos dois esquemas de tradução acima é exatamente a mesma linguagem, consistindo numa árvore gramatical com os mesmos operadores e operandos, porém mostra que as ações são feitas na ordem errada. As ações no esquema transformado de tradução terminam por último, pois corresponde a que serão realizadas exatamente antes do atributo ser calculado durante a análise sintática *bottom-up*.

Herdando Atributos na Pilha do Analisador Sintático

Um analisador sintático *bottom-up* reduz o lado direito de uma produção $A \rightarrow X_1 X_2 \ldots X_n$ através da remoção de X_1 e de X_2 do topo da pilha do analisador sintático, substituindo-os por A. Suponhamos que X_1 possua o atributo sintetizado $X_1.s$, o qual a implementação da Seção 5.3 armazene junto com X_1 na pilha do analisador sintático.

Uma vez que o valor de $X_1.s$ já esteja na pilha antes que quaisquer ações tenham lugar na redução de $X_1 X_2$ para A, este valor pode ser herdado por X_2. Isto é, se um atributo herdado $X_2.i$ for definido pela regra de cópia $X_2.i = X_1.s$, o valor de $X_1.s$ pode ser acessado onde $X_2.i$ for preciso. Como veremos, as regras de cópia desempenham um importante papel na avaliação de atributos herdados durante a análise sintática *bottom-up*.

Exemplo 5.17. O tipo de um identificador pode ser passado por uma regra de cópia utilizando-se atributos herdados, como mostrado na Fig. 5.31 (a) para a Fig. 5.31. Examinaremos primeiro os movimentos feitos por um analisador sintático *bottom-up* para a entrada

[illegible]

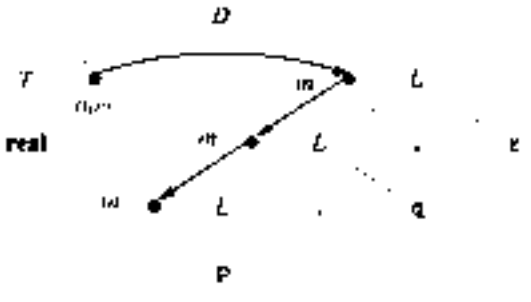


Fig. 5.32. Árvore para L, $L_1 = L_2$, $tipo$

ENTRADA	estado	PRODUÇÃO USADA
real [illegible]		
[illegible]	**real**	
[illegible]	T	$T \rightarrow$ **real**
[illegible]	T ,	
[illegible]	T , **id**	$L \rightarrow$ **id**
[illegible]	T L	
[illegible]	T L ,	
[illegible]	T L , **id**	$L \rightarrow L$, **id**
	T L	
	T L , ...	
	T L	$L \rightarrow L$, **id**
	D	$D \rightarrow T$ L

Fig. 5.33. Sequência de lado direito para L foi reduzido. Estado é o mesmo [illegible] de lado direito.

Em seguida, mostraremos como o valor do atributo $T.tipo$ pode ser acessado quando as produções para L forem aplicadas. Eis aqui o de tradução que descrevemos implementar a

$D \rightarrow$	T	{ $L.in := T.tipo$ }
	L	
$T \rightarrow$	**int**	{ $T.tipo := integer$ }
$T \rightarrow$	**real**	{ $T.tipo := real$ }
$L \rightarrow$		{ $L_1.in := L.in$ }
	L_1 , **id**	{ *declare_tipo*(**id**.*entrada*, $L.in$) }
$L \rightarrow$	**id**	{ *declare_tipo*(**id**.*entrada*, $L.in$) }

Se ignorarmos as ações no esquema de tradução acima, a sequência de movimentos realizados pelo analisador sintático para a entrada da Fig. 5.32 é exibida na Fig. 5.33. Para maior clareza, mostramos na pilha o símbolo gramatical correspondente em lugar do estado e o identificador efetivo em lugar de **id**.

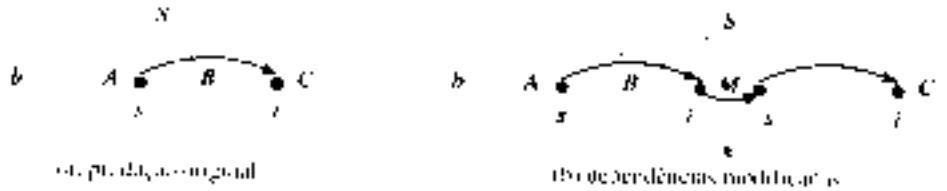
Suponhamos, como na Seção 5.5, que a pilha do analisador sintático seja implementada como um par de arrays *estado* e *val*. Se a entrada *estado*[*i*] estiver preenchida com o símbolo gramatical X, então *val*[*i*] deverá conter o valor do atributo sintetizado $X.s$. O conteúdo desses arrays é mostrado na Fig. 5.33. Note-se que a cada vez que o lado direito de uma produção para L for reduzido na Fig. 5.33, T estará na pilha exatamente abaixo desse lado direito. Podemos usar este fato para localizar o valor de atributo $T.tipo$.

A implementação da Fig. 5.34 usa função *declare_tipo* para colocar em local conhecido, relativo ao topo, na pilha *val*. Separamos o corpo em todos os da entrada de toda da pilha exatamente antes e após a redução ser feita (Seção 5.5), respectivamente. A partir das regras de cópia que definem L, sabemos que $T.tipo$ pode ser usado em lugar de $L.in$.

Quando a produção $L \rightarrow$ **id** for aplicada, **id** *entrada* estará no topo da pilha *val* e $T.tipo$ estará exatamente abaixo da mesma. Por conse-

PRODUÇÃO	COMPLEMENTO
$D \rightarrow T$,	
$T \rightarrow$ **int**	*val*[*ntopo*] := *integer*
$T \rightarrow$ **real**	*val*[*ntopo*] := *real*
$L \rightarrow L_1$, **id**	*declare_tipo*(*val*[*ntopo*], *val*[*ntopo*−1])
$L \rightarrow$ **id**	*declare_tipo*(*val*[*ntopo*], *val*[*ntopo*−1])

Fig. 5.34. Código de $T.tipo$ em [illegible] lugar de $L.in$



(a) Árvore de análise sintática original

(b) Árvore de dependências mostrando regras semânticas

Fig. 5.35. Árvore de análise sintática e árvore de dependências mostrando regras semânticas

gurar o valor de *A.s* antes de *B* ser reduzido a *Y*. Analogamente, uma ocorrência de *B.s* em uma ação deve ser substituída por *Y.s*. Ocasionalmente, uma ação pode ser associada a uma produção com mais de um símbolo *B*, caso em que cada ocorrência deve ser substituída por *Y.s*, que deve ser a última ocorrência de *B* na produção. Se o símbolo *B* não tiver atributo *s*, ou se *B* for um símbolo terminal, então *B.s* será substituído por *Y*. Se a ação for associada a uma produção com *A* no lado direito, então *A.s* será substituído por *X.s*. Se a ação for associada a uma produção com *A* na cabeça, então *A.s* será substituído por *Z.s*.

Simulando a Avaliação de Atributos Herdados

A pergunta na pilha é: o analisador sintático pode ter certeza de que o valor de atributo herdado de que uma gramática necessita para qualquer posição no valor de atributo seja sempre corretamente computado?

Exemplo 5.18. Como uma ilustração, vamos tentar predizer a posição em que colocaremos a seguinte regra de ação:

PRODUÇÃO	REGRAS SEMÂNTICAS	
$S \rightarrow aAB$	$C.i = A.s$	
$S \rightarrow bABC$	$C.i = A.s$	(5.9)
$C \rightarrow c$	$C.s = g(C.i)$	

O valor de atributo sintetizado *A.s* é fornecido por uma regra de cópia. Note se pode ou não colocar *B* entre *A* e *C* na pilha. Quando a redução através de $C \rightarrow c$ for realizada, o valor de *C.i* estará em *val*[*top*−1] ou em *val*[*top*−2], mais uma vez dependendo de qual produção estiver sendo usada.

No Fig. 5.35 (a), o nó-pai rotulado com *B* é irmão imediato à direita do nó-pai rotulado com *A*. Assim, *A.s* estará em *val*[*top*] quando a redução $B \rightarrow c$ for realizada. Se estivermos analisando sintaticamente de acordo com a produção $S \rightarrow bABC$, então *C.i* herdará o valor de *A.s*, que estará em *val*[*top*−1] quando a redução $C \rightarrow c$ for realizada. Se estivermos analisando de acordo com a produção $S \rightarrow aAB$, então *C.i* herdará o valor de *A.s*, que estará em *val*[*top*] quando a redução $C \rightarrow c$ for realizada.

No Fig. 5.35 (a), o nó-pai rotulado com *B* é irmão imediato à esquerda do nó-pai rotulado com *C*. Assim, *B.s* estará em *val*[*top*] quando a produção $C \rightarrow c$ for aplicada. Independentemente da primeira ou segunda produção estiver sendo usada, a seguinte modificação de 5.9:

PRODUÇÃO	REGRAS SEMÂNTICAS
$S \rightarrow aAB$	$C.i = A.s$
$S \rightarrow bABC$	$B.i = A.s$; $C.i = B.s$
$C \rightarrow c$	$C.s = g(C.i)$
$B \rightarrow e$	$B.s = B.i$

Não termos nos atendido em provar também se nós podemos simular regras semânticas que não sejam regras de cópia. Por exemplo, considere

PRODUÇÃO	REGRAS SEMÂNTICAS	
$S \rightarrow aA$	$C.i = 0.1 + 1$	(5.10)

Desta vez a regra que define *C.i* não é uma regra de cópia; em vez disso, o valor de *C.i* é calculado a partir de *A.s*. Este problema pode também ser resolvido usando-se a transformação

PRODUÇÃO	REGRAS SEMÂNTICAS	
$S \rightarrow aA$	$A.i = 1$; $C.i = A.s$	(5.11)
$A \rightarrow \epsilon$	$A.s = A.i + 1$	

O não-terminal *A* herda *A.i* através de uma regra de cópia. Seu atributo sintetizado *A.s* é então fornecido por uma regra de cópia. *C* herda então o valor de *A.s* através de uma regra de cópia. Assim, substituímos *A.s* por *C.i* em $C \rightarrow c$, encontramos o valor de *C.i* no lugar de 1 em *val*[*top*], e calculamos *C.s* = *C.i* + 1. Assim, substituímos *A.s* por *A.i* + 1 em $A \rightarrow \epsilon$, encontramos o valor de *A.i* em *val*[*top*], e calculamos *A.s* = *A.i* + 1. Desta vez não necessitamos de *C* nesses momentos intermediários, durante a redução de uma cadeia terminal para o não-terminal *C*, quando seu valor foi empurrado anteriormente na pilha juntamente com *A*.

Exemplo 5.19. Três não-terminais marcadores, *L*, *M* e *N*, são usados na Fig. 5.36 para assegurar que o valor de atributo herdado *B.in* apareça em uma posição calculável na pilha de analisador sintático empregado para sua avaliação. Os atributos originais aparecem na Fig. 5.22 e sua relevância para a formatação de texto é explicada no exemplo 5.15.

A inicialização é feita usando-se $L \rightarrow \epsilon$. A produção para S é $S \rightarrow L\ B$ na Fig. 5.36 e, consequentemente, *L* permanece em pilha, enquanto o subárvore abaixo de *B* for reduzida. O valor 10 do atributo herdado *B.in* = *L.s* é colocado na pilha durante a análise sintática pela regra $L \rightarrow \epsilon$ associada a $L \rightarrow \epsilon$.

O marcador *M* em $B \rightarrow B_1\ M\ B_2$ desempenha um papel similar àquele de *M* na Fig. 5.35, assegurando que o valor de *B.in* apareça exatamente abaixo de B_1 na pilha durante a análise sintática. Na produção $B \rightarrow$ **if** *E* **then** S **else** S, a união terminal, *UNION*, é usada como em (5.8). Herdada através da regra de cópia $S.s = B.in$, o valor de atributo do qual *B* independe sintaticamente, o valor de *B.in* pela regra $A.s = \textit{newpage}(M.s)$. A consequência desejada é que o valor de *B.in* esteja sempre imediatamente abaixo do lado direito, quando reduzimos para *B*.

Os fragmentos de código que implementam a definição dirigida pela sintaxe da Fig. 5.36 são mostrados na Fig. 5.37. Todos os atributos herdados são estabelecidos pelas regras de cópia da Fig. 5.36, e, por conseguinte, a implementação coloca seus valores corretamente em posições na pilha. Os nossos exemplos pequenos, todos cabem nos

PRODUÇÃO			REGRAS SEMÂNTICAS
S	$\rightarrow$	$L\ B$	$B.in = L.s$
			$S.s = B.s$
L	$\rightarrow$	ϵ	$L.s = 10$
B	$\rightarrow$	$B_1\ UB_2$	$B_1.in = B.in$
			$M.i = B.in$
			$B_2.in = M.s$
			$B.s = \textit{union}(B_1.s, B_2.s)$
$B \rightarrow B_1$ **sub** $N\ B_2$			$B_1.in = B.in$
			$N.i = B.in$
			$B_2.in = N.s$
			$B.s = \textit{down}(B_1.s, B_2.s)$
B	$\rightarrow$	**TEXTO**	$B.s = \textit{texto}(L.s, B.in)$
M	$\rightarrow$	ϵ	$M.s = B.i$
N	$\rightarrow$	ϵ	$N.s = \textit{reduzpont}(N.i)$

Fig. 5.36. Todos os atributos herdados são definidos através de regras de cópia

PRODUÇÃO	REGRA SEMÂNTICA
$S \rightarrow L\ B$	$val[top] = val[top]$
$L \rightarrow \lambda$	$val[top] = 10$
$B \rightarrow B\ 0\ B$	$val[top] = base \cdot val[top-2] + val[top]$
$B \rightarrow B\ 1\ B$	$val[top] = base \cdot val[top-2] + val[top]$
$B \rightarrow$ bexto	$val[top] = val[top] + val[top+1]$
$0 \rightarrow 0$	$val[top] = 0$
$1 \rightarrow 1$	$val[top] = 1$

Fig. 5.17 A representação da L-atribuição da gramática do Exemplo 5.30

valores nulos sobre o topo da pilha antes de iniciarmos uma redução, respectivamente.

Uma colocação sistemática de marcadores, como nas especificações *Yacc* e *Bison* (Seção 4.9), permite uma poderosa avaliação de definições L-atribuídas durante a análise sintática LR. Uma vez que existe somente uma produção para cada marcador, a ação a ser executada reconhece LL ou LR quando os marcadores são removidos. Qualquer gramática LL tem uma gramática LR, e quando os marcadores são colocados em gramáticas LL de tal forma que, conforme descrito acima, uma gramática LR é obtida, o esquema sintático pode ser executado durante a análise LR. Entretanto, esquemas sintáticos podem ser executados em muitas gramáticas LR, mas os marcadores não podem ser colocados em posições arbitrárias nessas gramáticas.

As idéias das seções precedentes podem ser formalizadas pelo seguinte algoritmo.

Algoritmo 5.18. Análise sintática e tradução *bottom-up* com atributos herdados.

Entrada. Uma definição L-atribuída com uma gramática LL subjacente.

Saída. Um analisador sintático que computa os valores de todos os atributos em sua pilha sintática.

Método. Vamos assumir por simplicidade que cada símbolo gramatical tem apenas um atributo herdado ou um sintetizado, mas não ambos. Se X for um símbolo terminal, o atributo sintetizado será o valor léxico retornado com X pelo analisador léxico. Lembre-se que o valor léxico é mantido na pilha, imediatamente abaixo, sempre que X estiver no topo da pilha.

Para cada produção $A \rightarrow X_1 X_2 \cdots X_n$ introduza n novos marcadores $M_1, M_2, \ldots, M_n$ e substitua a produção por $A \rightarrow M_1 X_1 M_2 X_2 \cdots M_n X_n$. Cada símbolo sintetizado X_i será colocado na pilha imediatamente abaixo de M_i. Assim, cada X_i do atributo herdado X_i se encontrará, sempre, em uma posição fixa relativa ao topo da pilha.

Uma variável importante é a que indica quanto abaixo do topo, no momento, está o atributo X_i de cima que será encontrado na posição de X_i em uma redução. Chamemos essa variável de $top\text{-}diff[i]$. Se a produção $A \rightarrow M_1 X_1 \cdots M_n X_n$ for reduzida, então $top\text{-}diff[n]$ é 0, $top\text{-}diff[n-1]$ é o comprimento do corpo de X_n, e, em geral, $top\text{-}diff[i]$ é a soma dos comprimentos de $X_{i+1}, \ldots, X_n$. Se o atributo herdado X_i for necessário para a computação de X, então ele estará em $val[top - (n-i) + 1]$. Se X_i estiver em $val[top - (n-i+2) + 1]$, estará em $val[top - (n-i) - 1]$, e assim por diante. Se X_i for um símbolo terminal, o atributo herdado estará em $val[top - (n-i)]$. Uma vez que tenhamos computado X_i, o valor de X_i estará em $val[top - (n-i) - 1]$, que se tornará o novo símbolo de topo da pilha após a redução. Note se que o valor de $top\text{-}diff[i]$ será importante em tempo de compilação, mas não precisamos estar certos de que estamos reduzindo a um não terminal particular para entender que não pode ser nossa tarefa descobrir os valores próprios em nossa cabeça em geral, que é o que queremos aplicar. Pedimos ao leitor para que verifique o fato de que uma gramática qualquer LL(1) com marcadores é ainda LR(1).

O segundo conjunto de ações que introduzimos a um símbolo com marcador digamos, uma ação da produção $A \rightarrow M_1 X_1 \cdots M_n X_n$. Conseqüentemente, temos somente que computar o atributo sintetizado X_i que se quer X_i for computado, e inseri-lo na posição da pilha exatamente abaixo da posição em que inserimos o próprio X_i. Os atributos necessários para computar X_i estão, naturalmente, disponíveis em posições calculáveis na pilha, a partir de posições dos valores X_i's durante a redução.

As seguintes simplificações reduzem o número de marcadores, e, segunda, em particular, evitam possíveis conflitos sintáticos nas gramáticas resultantes. Os esquemas

1. Se X_i não possuir atributo herdado, não precisamos utilizar o marcador M_i. Naturalmente, as posições esperadas para os atributos na pilha não mudam se M_i for omitido, mas essa mudança pode ser incorporada facilmente ao analisador sintático.
2. Se X_i é o símbolo mais à esquerda no corpo de uma produção, digamos $A \rightarrow X_1 \cdots X_n$, então podemos substituir M_i uma vez que sempre através de pesquisa através de $top\text{-}diff[1]$ para localizar onde o operando está abaixo de X_i na pilha. Esse valor pode subsequentemente ser usado para X_i igualmente.

Substituindo Atributos Herdados por Sintetizados

É possível, em muitas vezes se evitar o uso de atributos herdados através da reformulação da gramática. Considere, por exemplo, uma declaração em Pascal pode ser vista como uma lista de identificadores seguida por um tipo, por exemplo, `m, n, o : integer`. Uma gramática para tais declarações pode ter duas produções da forma

$$D \rightarrow L : T$$
$$T \rightarrow \textbf{integer} \mid \textbf{char}$$
$$L \rightarrow L, \textbf{id} \mid \textbf{id}$$

Como os identificadores são gerados por L, que é especificado estar à esquerda, não podemos associar o tipo a um identificador usando explicitamente os atributos sintetizados de forma semântica. Herdando um tipo à direita de L na primeira produção, temos uma declaração original pela sintaxe que não é L-atribuída e a tradução baseada na mesma não pode ser realizada durante a análise sintática.

Uma solução para esse problema está em reestruturar a gramática de forma a colocar o tipo como o último elemento da lista de identificadores.

$$D \rightarrow \textbf{id}\ L$$
$$L \rightarrow \textbf{id}, L \mid T$$
$$T \rightarrow \textbf{integer} \mid \textbf{char}$$

Agora, o tipo pode ser carregado para cada identificador sintetizado L seguinte. Assim, à medida que cada identificador está sendo gerado por L, seu tipo pode ser colocado na tabela de símbolos.

Uma Difícil Definição Dirigida pela Sintaxe

O Algoritmo 5.3 para implementar atributos herdados durante a análise sintática é bastante geral, se estendendo para algumas gramáticas LR. Mas não todas. A definição L-atribuída (Fig. 5.28) está baseada numa gramática

Produção			Regras Semânticas
A	$\rightarrow$	X	$L.conta = 1$
L_1	$\rightarrow$	X	$L_1.conta = L_2.conta + 1$
L	$\rightarrow$	ϵ	*imprima* $L.conta$

Fig. 5.38. Uma definição dirigida pela sintaxe.

para LR(1) simples, mas não pode ser implementada na forma descrita na Seção 5.4. O não-terminal Q em $Q \rightarrow \epsilon$ herda a contagem do número de X's pertencentes a P. A regra para $P \rightarrow QX$ é a parte difícil. Ao analisar sintaticamente ascendentemente, através de uma pilha, aquela ação pode saber o número de X's na entrada.

5.7 AVALIADORES RECURSIVOS

Funções recursivas, que avaliam atributos para cada nó de uma árvore sintática, podem ser construídas a partir de uma definição dirigida pela sintaxe usando uma generalização das técnicas para tradução preditiva na Seção 5.5. Tais funções nos permitem implementar definições dirigidas pela sintaxe que não podem ser implementadas simultaneamente com a análise sintática. Nesta seção, nós introduzimos uma única função de tradução recursiva descendente. A função visita os filhos de um nó para um não-terminal em uma ordem determinada pela produção para aquele nó, mas é preciso que os filhos sejam visitados numa ordem da esquerda para a direita. Na Seção 5.10, veremos uma técnica para o caso em que mais de uma passagem pode ser necessária, através da associação de múltiplos procedimentos aos não-terminais.

Travessias da Esquerda para a Direita

No Algoritmo 5.2, mostramos como a tradução dirigida pela sintaxe de uma gramática LL(1) pode ser implementada através de um procedimento recursivo para cada não-terminal. Todas as definições L-atribuídas, dirigidas pela sintaxe, podem ser implementadas com uma função recursiva similar para cada não-terminal da gramática. Através do exame da produção aplicada, a função pode determinar aqueles são seus filhos. A função para um não-terminal A recebe como argumentos os valores dos atributos herdados para A como argumentos e como resultado retorna os valores dos atributos sintetizados para A.

Os detalhes da construção são exatamente como no Algoritmo 5.2, exceto no passo 2, onde a função para um não-terminal decide que produção usar baseada no símbolo corrente de entrada. A função neste ponto emprega unicamente $select$ para determinar a produção usada a um nó. Damos um exemplo para ilustrar o método.

Exemplo 5.20. Consideremos a definição dirigida pela sintaxe para determinar o tamanho e a largura de caixas da Fig. 5.22. A não-terminal B possui um atributo herdado ps e um atributo sintetizado h. Usando o Algoritmo 5.2, modificado como mencionado acima, escrevemos a função para B mostrada na Fig. 5.39.

A função B tem como argumentos um nó e avalia o comprimento correspondente a ps daquele nó e retorna o valor correspondente a $B.ht$ do nó. A função possui um enunciado $case$ para cada produção com B à esquerda. O código correspondente a cada produção soma as regras semânticas associadas à produção. A ordem na qual as regras são

```
função B(n: tipo
    var ps1, ps2, ht1, ht2;
início
    caso a produção em n seja
    'B → B B.1':
        ps1 := ps;
        ht1 := B(filho(n, 1), ps1);
        ps2 := B(filho(n, 2), ps2);
        retornar max(ht1, ht2)
    'B → B sub B.2':
        ps1 := ps;
        ht1 := B(filho(n, 1), ps1);
        ps2 := 0.8 * ps;
        ht2 := B(filho(n, 3), ps2);
        retornar desloc(h, ht1, ht2)
    'B → texto':
        retornar ps * comprimento(n)
    default:
        erro
    fim
fim
```

Fig. 5.39. Função para o não-terminal B da Fig. 5.22.

aplicadas precisa ser tal que os atributos herdados de um não-terminal sejam computados antes que a função para o não-terminal seja chamada.

No código correspondente à produção $B \rightarrow B$ **sub** B, as variáveis ps, $ps1$ e $ps2$ guardam os valores dos atributos herdados de B, de B_1 e de B_2, respectivamente. ht_1 e ht_2 guardam os valores de $B_1.ht$ e $B_2.ht$. Usamos a função $desloc(n, x, y)$ para retornar o resultado da nó n. Como B_1 é o primeiro dos três filhos do nó n, o valor de $B_1.ht$ é determinado pela chamada de função $B(filho(n, 1), ps)$.

Outras Travessias

Uma vez que uma árvore gramatical explícita está disponível, estamos livres para visitar os filhos de um nó em qualquer ordem. Consideremos a definição, não L-atribuída, do Exemplo 5.21. Normalmente as definições dirigidas pela sintaxe precisam ser visitadas da esquerda para a direita, enquanto que os filhos de um nó para a outra produção precisam ser visitados da direita para a esquerda.

Produção			Regras Semânticas
A	$\rightarrow$	$L\ M$	$L.i := f(A.i)$ $M.i := g(L.s)$ $A.s := f(M.s)$
A	$\rightarrow$	$Q\ R$	$R.i := r(A.i)$ $Q.i := q(R.s)$ $A.s := f(Q.s)$

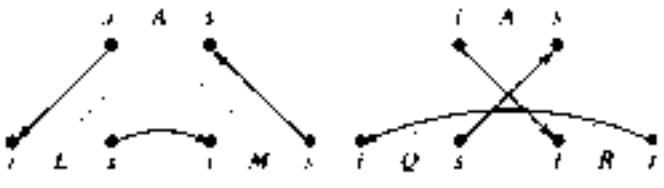


Fig. 5.40. Produções e regras semânticas para árvores sintáticas.

```
função A(n, a);
início
    caso a produção em n seja
      1 — A, M, Q, R: {calcula a(n) usando a(A), a(M), a(Q) e a(R)
          b = a(n);
          A(b, a(b));
          m(b) = a(b);
          a(n) = M(b, a(b), a(n));
          retornar a(n);
      3 — Q R: {calcula a(n) usando a(Q) e a(R)
          a(Q) = a(n);
          a(n) = M(a(n), a(n));
          q(n) = q(n);
          q(n) = Q(a(n), q(n));
          retornar q(n);
      default:
          {erro}
      fim
fim;
```

Fig. 5.41. Esqueleto da função para as dependências na Fig. 5.40 (determinando todos os valores de atributos).

O exemplo abstrato ilustra o processo de uso de funções como *M* e *Q* para calcular os atributos dos nós de uma árvore sintática. As funções não precisam depender de ordem na qual os nós da árvore são visitados. Duas considerações principais para a escolha dessa ordem são a necessidade de que os atributos herdados de um nó sejam computados antes que o nó seja visitado pela primeira vez e a necessidade de que os atributos sintetizados sejam computados antes que eles sejam usados pela última vez.

Exemplo 5.21. Cada um dos não terminais na Fig. 5.37 possui um atributo herdado e um sintetizado. Os grafos de dependências para as duas produções também são mostrados. As regras associadas a $A \rightarrow LM$ estabelecem dependências da esquerda para a direita enquanto que as associadas a $A \rightarrow QR$ estabelecem uma da direita para a esquerda.

A função para um nó terminal é a mesma da Fig. 5.41. Assumimos que as funções para *L*, *M*, *Q* e *R* possam ser construídas. As regras na Fig. 5.41 são semelhantes àquelas que não terminais e seus atributos, por exemplo, *b* e *c* são os atributos correspondentes a *L* e *Q*.

O código correspondente à produção $A \rightarrow LM$ é construído como no Exemplo 5.20, isto é, determina o atributo herdado de *L* chamando a função para *L*. Uma vez determinado o atributo sintetizado de *M*, retornamos o processo para *M*. O código correspondente a $A \rightarrow QR$ visita a subárvore para *R* antes que visite a subárvore para *Q*. Novamente, a ordem para as duas produções é mostrada [...]

PRODUÇÃO	REGRAS
$D \rightarrow T\ I$	$I.tipo = T.tipo$
$T \rightarrow$ **int**	$T.tipo = inteiro$
$T \rightarrow$ **real**	$T.tipo = real$
$I \rightarrow I_1, I_2$	$I_1.tipo = I.tipo$ $I_2.tipo = I.tipo$
$I \rightarrow I$	$I.tipo = I.tipo$
$I \rightarrow I$ [**num**]	$I.tipo = array(\text{**num**}.valor, I.tipo)$
$I \rightarrow$ **id**	$adicionaTipo(\text{**id**}.entrada, I.tipo)$

Fig. 5.42. Possíveis tipos para os identificadores numa declaração.

5.8 ESPAÇO PARA OS VALORES DE ATRIBUTOS EM TEMPO DE COMPILAÇÃO

Nesta seção, consideramos novas atribuições de espaço em tempo de compilação para os valores de atributos. Iremos usar as informações providas pelo grafo de dependências para uma árvore gramatical e, posteriormente, esta seção é dedicada a considerações que, utilizando diversas gramáticas, determinam a ordem de avaliação a partir dos grafos de dependências. Na próxima seção, consideramos o caso em que a ordem de avaliação pode ser pré-estabelecida independentemente de árvores sintáticas particulares e discutimos a respeito do espaço para os atributos mínimo e para todo o tempo que um valor com [...] for armazenado.

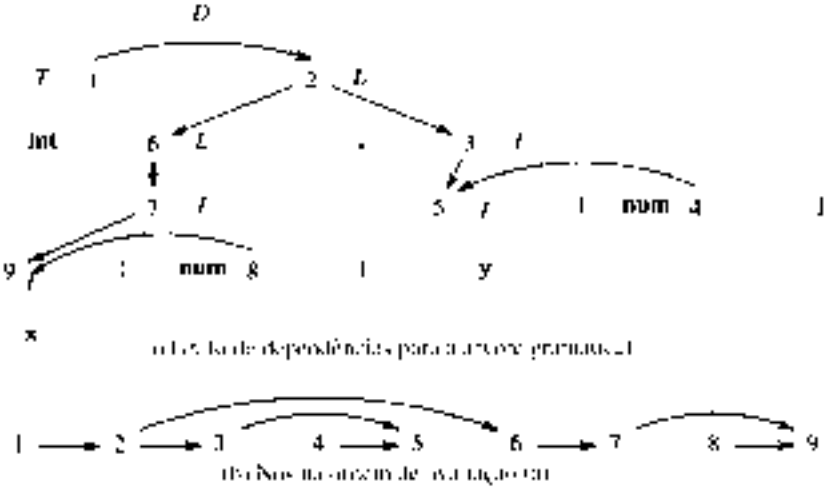
Nada nos obriga a avaliação para os atributos a ser executada explicitamente de pesquisa em profundidade. O tempo de vida de um atributo começa quando o nó é pela primeira vez criado e termina quando todos os atributos que dele dependem tenham sido computados. Podemos economizar espaço gerenciando os objetos de um atributo somente durante seu tempo de vida.

Para calcular atributos nesta seção, aplicamos a qualquer ordem de avaliação, mesmo considerando a regra de função dirigida pela sintaxe, [...] por exemplo, para manter a informação de tipo de dados dos identificadores numa declaração.

Exemplo 5.22. A definição dirigida pela sintaxe na Fig. 5.42 é uma extensão daquela da Fig. 5.4 a qual permite declarações da forma

[...]	:	5.01
[...]	:	15.101

Uma árvore gramatical para (5.101) é mostrada pelas linhas pontilhadas na Fig. 5.43(a). Os números dos nós são descritos ao próximo exemplo.



(a) Grafo de dependências para a árvore gramatical

(b) Nós na ordem de avaliação (1)

Fig. 5.43. Exemplo de dependências e ordens sobre os atributos.

```
para cada nó n_1, n_2, ..., n_k faça início
    para cada um dos tempos de vida terminados com a avaliação do nó faça
        desocupar o registrador do
    se algum registrador r estiver marcado então início
        descarregar r;
        atribuir n ao registrador r;
        retornar os registradores marcados para o pool
    fim
    se não (n é um registrador livre que pode ser usado)
        pegar um registrador previamente de pool
    /* a variável escolhida, se houver, poderá ser inserida aqui */
    se houver tempo de vida de n terminando então
        retornar o registrador de n para o pool
fim
```

Fig. 5.44. Atribuindo valores a atributos sintetizados.

plo. Como na Fig. 5.43, o produtor apropriado de T_i é herdado por cima a partir do nó rotulado abaixo para os identificadores na declaração. O valor da parte de T type para L em $L \to T$ type L_1 depende de T type. A definição dirigida pela sintaxe na Fig. 5.12 não é bem-fundada porque L não depende de nenhum valor de uma variável diferente de L_1 em $L \to L_1$ [num].

Reservando Espaço para os Atributos em Tempo de Compilação

Suponhamos que seja dada uma sequência de registradores para guardar valores de atributos. Para conveniência, assumimos que cada registrador possa guardar qualquer valor de atributo. Se os atributos representam tipos diferentes, podemos formar grupos de atributos que ocupem a mesma memória e considerar cada grupo separadamente. Usamos as informações sobre os tempos de vida dos atributos para determinar os registradores nos quais serão avaliados.

Exemplo 5.23. Suponhamos que os atributos sejam avaliados na ordem dada pelos números de nós no grafo de dependências da Fig. 5.43, construído no último exemplo. O tempo de vida de cada nó começa quando o atributo for avaliado e termina quando for usado pela última vez. Por exemplo, o tempo de vida do 1 termina quando o 2 for avaliado, porque 2 é o único nó que depende de 1. O tempo de vida de 2 termina quando 6 for avaliado.

Um método para avaliar os atributos que usa um pouco os registradores quanto possível é determinado na Fig. 5.45. Consideramos os nós do grafo de dependências D para uma árvore gramatical na ordem em que devem ser avaliados. Inicialmente, temos um pool de registradores $r_1, r_2, \ldots$. Se o atributo b for definido pela regra semântica $a = f(b_1, b_2, \ldots, b_k)$, e o tempo de vida de um ou mais dos $b_1, b_2, \ldots, b_k$ puder terminar com a avaliação de b, os registradores que guardam tais atributos são retornados ao pool após o serem usados. Sempre que possível, é avaliado um registrador que detinha um dos $b_1, b_2, \ldots, b_k$.

Os registradores usados durante a avaliação do grafo de dependências da Fig. 5.43 são mostrados na Fig. 5.45. Começamos pela avaliação do nó 1, registrador r_1. O tempo de vida do nó 1 termina quando o 2 for avaliado e, por conseguinte, 2 é avaliado em r_1. O nó 3 contém um registrador novo, r_2, porque o nó 4 ainda necessita do valor de 1.

Evitando Cópias

Podemos melhorar o método da Fig. 5.44 incluindo as regras de cópia como um caso especial. Uma regra de cópia possui a forma $A = B$, e, consequentemente, se o valor de B estiver em um registrador, o valor de B já aparece em A. O número de atributos definidos pelas regras de cópia pode ser significativo e, dessa forma, desejamos evitar a realização de cópias explícitas.

Um exemplo de nós gerados mesmo em uma forma ainda classe de equivalência. O método da Fig. 5.44 pode ser modificado como se segue. Em vez de alocar o valor de uma classe de equivalência em um registrador. Quando uma nova classe é considerada, verificamos primeiro se esta é definida por alguma regra de cópia. Se estiver, será vista já estará em um registrador e só vamos a classe de equivalência com valores naquele registrador. Sobretudo, um registrador é reservado *on-the-fly* somente no fim dos tempos de vida de todos os nós com valores naquele registrador.

Exemplo 5.24. O grafo de dependências na Fig. 5.43 é redesenhado na Fig. 5.46, com um conjunto de regras cópia definido por uma regra de cópia. A partir da definição dirigida pela sintaxe da Fig. 5.42, encontramos que o tipo determinado em cada elemento na lista de identificadores, resultando em que os nós 2, 3, 6 e 7 da Fig. 5.43 sejam cópias de 1.

Uma vez que 2 e 3 são cópias de 1, seus valores são obtidos a partir do registrador r_1 na Fig. 5.46. Note se o tempo de vida de 3 terminar quando 5 for avaliado, então o registrador r_2 que abrigava o valor de 3 não é retornado ao *pool* porque o tempo de vida de 2 na sua classe de equivalência não terminou ainda.

O seguinte código mostra como a declaração (5.1) do Exemplo 5.22 poderia ser processada por um compilador:

```
r_1 = mktable(...)          /* avaliação do nó 1, 2, 3, 6 e 7 */
r_2 = 8                     /* avaliação do nó 4 */
r_1 = enter(r_1, 'i', ...)  /* nó 5 */
enum_type(r_2)              /* nó 8 */
r_2 = 3.                    /* avaliação do nó 9 */
r_1 = enter(r_1, 'x', r_2)  /* nó 10 */
anchor_free(r_1, r_2).
```

Acima, r_1 é apontador para o início da tabela de símbolos para i e x, e os procedimentos *mktable*, *enter* e *enum_type* podem ser chamados nos pontos apropriados de forma a alocar os tipos de i e de x às suas entradas na tabela de símbolos.

5.9 RESERVA DE ESPAÇO EM TEMPO DE CONSTRUÇÃO DO COMPILADOR

Apesar de ser possível guardar valores numa única pilha durante uma travessia, podemos, algumas vezes, evitar a realização de cópias através da utilização de múltiplas pilhas. Em geral, se os dependentes entre os atributos tornam inconveniente colocar certos valores de cópia numa pilha, pode-nos ser mais natural criar uma sintaxe explicitamente consistente

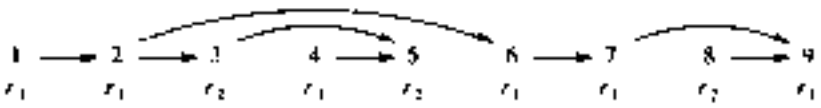


Fig. 5.45. Registradores usados para os valores de atributos na Fig. 5.43.

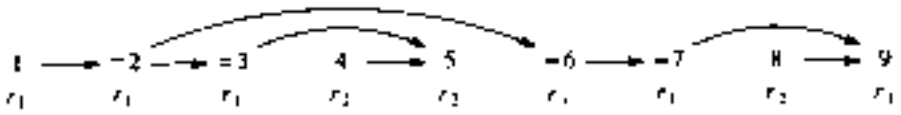


Fig. 5.46. Registradores usados ao avaliar em ordem as regras de cópia.

Já examinamos o uso de uma pilha para guardar valores de atributos durante a análise sintática *bottom-up* — Seções 5.3 e 5.4. Uma tradução orientada pela sintaxe também pode ser feita durante a análise descendente recursiva. Para esse caso, basta usar uma pilha de procedimentos, esse tema será discutido no Capítulo 7.

Outra questão pode ser calculada com outras técnicas para economizar espaço. As ações de impressão usadas extensivamente nos esquemas de tradução do Capítulo 2 usam atributos com valores de tipo cadeia de caracteres para, em tempo de execução, produzir sempre que possível. Em muitas construções sintáticas na Seção 5.2, possíveis apontadores para os nós, em vez de subárvores completas. Em geral, através de passar objetos extensos, podemos ganhar espaço mantendo apontadores para os mesmos. Essas técnicas serão aplicadas nos Exemplos 5.27 e 5.28.

Predizendo os Tempos de Vida a Partir da Gramática

Quando a ordem de avaliação para os atributos é obtida a partir de uma árvore gramatical, podemos predizer os tempos de vida dos atributos em tempo de construção do compilador. Por exemplo, suponhamos que os filhos sejam visitados da esquerda para direita durante uma travessia em profundidade como na Seção 5.6. Começando pelo nó para a produção $A \to B\ C$, a subárvore para B é visitada, a subárvore para C é visitada e retornamos ao nó para A. O nó para A não pode ser avaliado até que os atributos de B e de C estejam disponíveis. Generalizando, os tempos de vida podem ser determinados quando tivermos uma regra para A. Note que essas observações estão baseadas na produção $A \to B\ C$ e no fato de que, para essa produção, os nós para B e C são visitados. Nada pode ser dito a respeito das subárvores em B e C.

Com qualquer ordem de avaliação, o tempo de vida do atributo b associado com um nó B é, então, o valor de b pode ser guardado numa pilha durante o tempo de b. Aqui, b é um atributo com o mesmo não-terminal. Para a produção $A \to B\ C$, podemos usar uma pilha durante uma travessia em profundidade da seguinte forma:

Começamos pelo nó para A com os atributos herdados de A que já estão na pilha. Em seguida, avaliamos e empilhamos os valores de atributos herdados de B. Esses atributos permanecem na pilha à medida que avaliamos a subárvore de B, mas também os atributos sintetizados de B acima deles. Este processo é repetido com C, isto é, empilhamos seus atributos herdados, percorremos sua subárvore e retornamos com seus atributos sintetizados no topo. Usando $H(X,a)$ e $S(X,b)$ para os atributos herdados e sintetizados de X, respectivamente, a pilha agora contém

$$H(A,a),\ H(B,c),\ S(B,d),\ H(C,e),\ S(C,f) \qquad (5.11)$$

Todos os valores de atributos necessitados para computar os atributos sintetizados de A estão agora na pilha, e podemos, então, recortar para A com a pilha contendo

$$H(A,a),\ S(A,g)$$

Note se que o número de presumíveis elementos na pilha dos atributos herdados e sintetizados de um símbolo gramatical é fixo. Por conseguinte, em cada passo do processo, sabemos exatamente quantos registradores ou posições de memória precisamos para encontrar um atributo.

Exemplo 5.25. Suponhamos que os valores de atributos para a tradução de composição de tipos do Exemplo 5.22 sejam guardados numa pilha, como descrito acima. Começando pelo nó para a produção $B \to B\ B$ com $B_1 y$ no topo da pilha, os conteúdos da pilha antes e após a visita a um nó são mostrados na Fig. 5.47 à esquerda e à direita do nó, respectivamente. Como de praxe, a pilha cresce para baixo.

Note se que, exatamente antes de um nó para um não-terminal B ser visitado pela primeira vez, seu atributo q está no topo da pilha. Imediatamente antes da última visita, isto é, quando o ponteiro se dirige de volta para aquele nó, seus atributos s e q estão nas duas posições no topo da pilha. □

Quando um atributo b é definido por uma regra de cópia $B = C$ e o valor de c está no topo da pilha de valores de atributos, podemos necessitar empilhar uma cópia de c na pilha do analisador sintático. Podem haver mais oportunidades para se eliminar regras de cópia se mais de uma pilha for usada para abrigar valores de atributos. No próximo exemplo, usamos pilhas distintas para os atributos sintetizados e herdados. Uma comparação com o Exemplo 5.25 nos mostra que regras de cópia adicionais podem ser eliminadas se forem usadas pilhas separadas.

Exemplo 5.26. Com a definição dirigida pela sintaxe da Fig. 5.22, suponhamos que usemos pilhas separadas para os atributos herdados e para atributos sintetizados e mantenhamos as pilhas de tal forma que B e

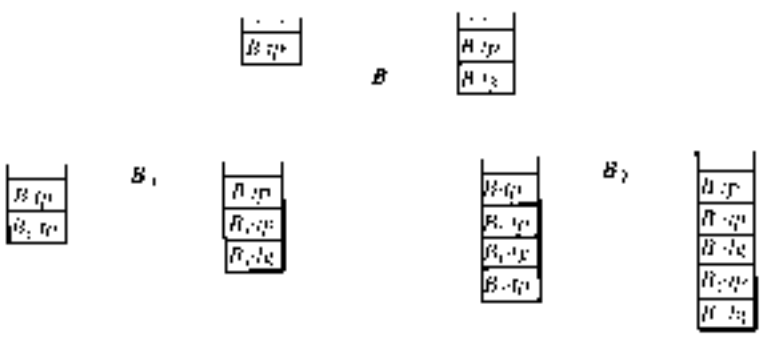


Fig. 5.47. Conteúdo da pilha antes e após a visita a um nó.

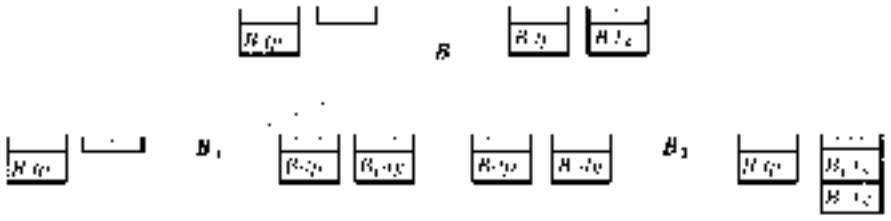


Fig. 5.48. Uso de pilhas separadas para os atributos herdados.

estará no topo da pilha de q exatamente antes de B ser primeiramente visitado e estará após ser pela última vez visitado. $B_1.qp$ estará no topo da pilha para q exatamente antes de B ser visitado.

Com pilhas separadas, podemos nos beneficiar de atributos nas regras de cópia $B.qp := B_1.qp$ e $B_2.qp := B.qp$ associadas a $B \rightarrow B\ B_2$. Como mostrado na Fig. 5.48, não precisamos empilhar $B_1.qp$, porque seu valor já está no topo da pilha como $B.qp$.

Um esquema de tradução baseado na definição dirigida pela sintaxe da Fig. 5.22 é mostrado na Fig. 5.49. A operação *empilhar*(v, p) empilha o valor v na pilha p, e *desempilhar*(p) desempilha o valor no topo da pilha p. Usamos *topo*(p) para referenciar o valor do elemento do topo da pilha p.

O próximo exemplo combina o uso de uma pilha para valores de atributos com ações para gerar código.

Exemplo 5.27. Aqui, consideramos as técnicas para implementar uma definição dirigida pela sintaxe especificando a geração de código intermediário. O valor de uma expressão *booleana* E é $E.t$ ou $E.f$, dependendo se E for falso. Em C, a subexpressão C não precisa ser avaliada se C for falso. A avaliação de uma tal expressão *booleana* é ensaiada na Seção 8.4.

As expressões *booleanas* na definição dirigida pela sintaxe na Fig. 5.50 são construídas a partir de identificadores e dos operadores **or**. Cada expressão C herda dois rótulos, $C.t$ e $C.f$, marcando os pontos para os quais o controle flui para o rótulo herdado $E.t$ ou $E.f$, se C for verdadeiro ou falso, respectivamente.

Suponhamos que $E \rightarrow E_1$ **or** E_2. Se E_1 é avaliado como verdadeiro, o controle flui para o rótulo herdado $E.t$, pois, caso contrário, E_1 é avaliado como verdadeiro e o controle flui para o código que avalia E_2. Um esboço do código gerado pela função *construir* mostra o início do código para E_1. As instruções condicionais são tratadas usando *gerar*. Para uma discussão posterior da relevância da Fig. 5.50 para a geração de código intermediário ver a Seção 8.4.

A definição dirigida pela sintaxe da Fig. 5.50 é L-atribuída e, por conseguinte, podemos construir um esquema de tradução para a mesma. O esquema de tradução da Fig. 5.51 usa um procedimento *emitir* de forma a gerar e emitir instruções incrementalmente. Podem ser mostradas na figura várias ações para o estabelecimento dos valores dos atributos herdados, inseridos antes dos símbolos gramaticais apropriados, como discutido na Seção 5.4.

$S \rightarrow$		{*empilhar*(*v*, *ip*)}
	M	
$B \rightarrow$	B_1	
	B	{*t*1 := *novo rótulo*(); *t*2 := *topo*(*ip*); *desempilhar*(*ip*); *empilhar*(*novo*(), *t*1, *t*2, *ip*)}
$M \rightarrow$	B_1	
	sub	{*empilhar*(*v*, *empilhar*(*novo*(), *ip*))}
	B_2	{*desempilhar*(*ip*); *t*1 := *topo*(*ip*); *desempilhar*(*ip*); *empilhar*(*novo*(), *t*1, *t*2, *ip*)}
$B \rightarrow$	**texto**	{*empilhar*(**texto**.*lexval*, *ip*)}

Fig. 5.49. Esquema de tradução com pilhas separadas.

O esquema de tradução da Fig. 5.52 usa duas pilhas separadas para guardar os valores dos atributos herdados $A.i$ e $E.i$. Como no Exemplo 5.26, as regras de cópia não possuem ações sobre as pilhas. Para implementar a regra $E \rightarrow$ **not** E_1, um novo rótulo é empilhado na pilha *ir_para* antes de E_1 ser visitado. O terminal **not** é dado uma tradução vazia, como E corresponde a *ir_para* $E.f$ e E corresponde a *ir_para* $E.t$, e consequentemente a pilha *ir_para* não muda neste exemplo, mas é necessária quando o operador **not** é adicionado à linguagem juntamente com o operador **or**. □

Tempos de Vida Não-Superpostos

Um registrador é uma forma especial de pilha. Se cada operação de empilhar for seguida por uma operação de desempilhar, podemos imaginar todos os dados em um único elemento da pilha. Nesse caso podemos usar um registrador no lugar de uma pilha. Em termos de tempos de vida, se os tempos de vida de dois atributos não se superpuserem, seus valores podem ser guardados no mesmo registrador.

Exemplo 5.28. A definição dirigida pela sintaxe da Fig. 5.36 constrói árvores sintáticas para expressões do tipo lista, com operadores de dois níveis de precedência. É obtida a partir do esquema de tradução da Fig. 5.38.

Alternativas para o tempo de vida de cada atributo de X ocorrem quando o atributo que depende dele é avaliado. Podemos mostrar que para qualquer árvore gramatical, os atributos de X podem ser armazenados no mesmo registrador. A seguinte argumentação explica aquela necessidade para análises gramaticais. A tradução é feita no tipo da subárvore com raiz X, no fragmento de árvore gramatical da Fig. 5.53.

A menor subárvore é obtida se $B \rightarrow c$ for aplicada, caso em que $B \rightarrow c$ é uma cópia de $B.t$ e, consequentemente, os dois valores estão juntamente. Para uma subárvore maior, a tradução à raiz da subárvore terá que ser para $X \rightarrow$ **op_aditivo** $I\ B_1$. O tempo de vida de $B.t$ termina quando $B_1.t$ é avaliado e, então, $B.t$ pode ser avaliado no mesmo registrador. A parte da hipótese indutiva de todos os atributos para instâncias do não terminal B na subárvore raiz para B_1 podem ser armazenados no mesmo registrador. Finalmente, $X.t$ é uma cópia de $B_1.t$ e, consequentemente, seu valor já está aí.

Regras Semânticas	
$E \rightarrow E_1$ **and** E_2	$E_1.i$ = *novo rótulo*() $E_2.i$ = $E.t$ $E.t$ = $E_2.t$ $E.f$ = $E.f$ $E.código$ = $E_1.código$ || *gerar*('**rótulo**' $E_1.i$) || $E_2.código$
$E \rightarrow$ **id**	$E.código$ = *gerar*('**se**' **id**.*lexval* '**ir_para**' $E.t$) || *gerar*('**ir_para**' $E.f$)

Fig. 5.50. Ações para a geração de código intermediário.

```
E →      E1.h = criar_nodo();
         E2.h = E.h;
         E1;
         E2;
         criar_filho(E.h, E1.s);
         E.s = E2.s;
E →      id   criar_folha(id.lexema); ir_para
         E.s =
         criar_ir_para(E.h);
```

Fig. 5.51. Tradução descendente para expressões de nodos.

Os cálculos de tradução na Fig. 5.51 usam os atributos herdados e sintetizados da Fig. 5.26, usando-se registadores para guardar os valores dos atributos $R.i$ e $R.s$ para todas as instâncias do não-terminal R.

Para completar a construção na Fig. 5.50, o código para a implementação do esquema acima é construído de acordo com o Algoritmo 5.21. O não-terminal R já não possui mais atributos, e A foi transformado em procedimento em lugar de uma função. A variável r foi tornada uma função f passível de ser chamada, sempre que R seja chamado, pois se não for necessário fazê-lo no esquema da Fig. 5.50. Este algoritmo pode ser aprimorado ainda mais eliminando-se a variável p e substituindo-se a chamada semântica de R pelo procedimento resultante, como na Seção 2.5. □

5.10 ANÁLISE DE DEFINIÇÕES DIRIGIDAS PELA SINTAXE

Na Seção 5.7, os atributos eram avaliados durante uma travessia de uma árvore sintática, com o emprego de funções mutuamente recursivas. As funções para um não-terminal A operam sobre os valores dos atributos herdados em A e nos valores dos atributos sintetizados daquele nodo.

O critério da Seção 5.7 se estende a tradutores que não podem ser realizados durante uma única travessia em profundidade. Aqui, novas funções são uma função separada para cada atributo sintetizado de cada não-terminal. Apesar de grupos de atributos sintetizados poderem ser calculados por uma única função. A construção na Seção 5.7 lida com os casos especiais em que todos os atributos sintetizados formam um grupo. O agrupamento dos atributos é determinado a partir das regras semânticas numa definição dirigida pela sintaxe. O algoritmo explicado adiante constrói a construção de um avaliador recursivo.

Exemplo 5.29. A definição dirigida pela sintaxe na Fig. 5.52 é motivada por um problema que iremos considerar no Capítulo 6. Resumidamente, o problema é como segue. Um identificador sobrecarregado pode ter um conjunto de possíveis tipos como resultado, uma expressão pode ter um conjunto de tipos possíveis. Informações contextuais são usadas para selecionar um dos possíveis tipos para cada subexpressão. O problema pode ser resolvido em duas passagens, uma passagem *bottom-up* para sintetizar o conjunto de tipos possíveis, seguida de uma passagem *top-down* para atribuir o conjunto a um único tipo.

As regras semânticas na Fig. 5.52 são uma abstração desse problema. O atributo sintetizado s na Fig. 5.52 representa o conjunto de

```
E1 →     E1.i = E.i;
         E2.i = E.i;
         E.s = f(E1.s, E2.s);
E →      id   E.s = if E.i in lexval(id.lexema) then
                    {lexval(id.lexema)}
                else {};
```

Fig. 5.52. Uma definição dirigida pela sintaxe.

Produção	Regras Semânticas
$T \rightarrow T\ R$	$R.i := T.tipo$ $E.tipo := R.s$
$R \rightarrow$ op_aditivo $T\ R_1$	$R.i := criar\_nodo(\text{op\_aditivo}.lexema, R.i, T.tipo)$ $R.s := R_1.s$
$R \rightarrow \epsilon$	$R.s := R.i$
$T \rightarrow$ num	$T.tipo := criar\_folha(\text{num}.tipo, \text{num}.val)$

Fig. 5.53. Uma definição dirigida pela sintaxe adaptada a partir da Fig. 5.26.

possíveis tipos do atributo sintetizado, a informação contextual. Um atributo adicional sintetizado t que não pode ser avaliado na mesma passagem que s, poderia representar o código gerado ou o tipo selecionado para uma subexpressão. Os grafos de dependências para as produções na Fig. 5.53 são mostrados na Fig. 5.58. □

Avaliação Recursiva de Atributos

O grafo de dependências para uma árvore gramatical é formado encaixando-se grafos menores correspondentes às regras semânticas para uma produção. O grafo de dependências D para a produção p é baseado somente nas regras semânticas para uma única produção, isto é, nas regras semânticas para os atributos sintetizados do lado esquerdo de p e para os atributos herdados dos símbolos gramaticais do lado direito da produção. Isto é, o grafo D mostra as dependências locais somente. Por exemplo, todos os nodos do grafo de dependências para $E \rightarrow E\ E_1$ na Fig. 5.58 são entre instâncias do mesmo atributo. A partir de um grafo de dependências não podemos dizer se os atributos precisam ser computados antes de outros atributos.

Um exame mais apurado do grafo de dependências para a árvore gramatical da Fig. 5.59 mostra que os atributos de cada instância do não-terminal E precisa ser avaliada na ordem $E.s$, $E.i$, $E.t$. Note-se que todos os atributos da Fig. 5.59 podem ser avaliados em três passagens, uma passagem *bottom-up* para avaliar os atributos s, uma passagem *top-down* para avaliar os atributos i, e uma passagem *bottom-up* final para avaliar os atributos t.

Num avaliador recursivo, a função para um atributo sintetizado b de A calcula os valores de alguns dos atributos recebidos como parâmetros. Em geral, se o atributo sintetizado b de A pode depender do atributo herdado c de A, então a função para $A.b$ tomará $A.c$ como parâmetro. Antes de analisarmos as dependências, consideramos um exemplo mostrando seus usos.

Exemplo 5.30. As funções $E.s$ e $E.t$ na Fig. 5.60 retornam os valores dos atributos sintetizados s e t a um nó n rotulado E. Como na Seção 5.7, existe um procedimento avaliador para cada produção na função para um não-terminal. O código executado em cada chamada desse simula as regras semânticas associadas à produção na Fig. 5.57.

Da discussão acima a respeito do grafo de dependências na Fig. 5.58 sabemos que o atributo i de um nó numa árvore gramatical pode depender de t. Consequentemente, passamos o atributo herdado i como parâmetro para a função $E.t$ que é associada ao atributo t. Como

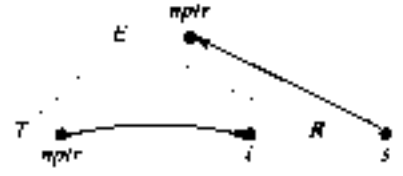


Fig. 5.54. Grafo de dependências para $E \rightarrow T\ R$

```
E → T        { t = T.nova; E.nova = t; agora contém R = t }
    R        { E.nova = R.nova; R contém t como R.herdado }
R → opaditivo T
                { t = novo nó op-aditivo; t.esquerdo = R.herdado;
                  t.direito = T.nova; R.nova = t }
    R'
R → ε
T → num { T.nova = criar_folha(num, num.val) }
```

Fig. 5.55. Esquema de tradução com ações para construir árvores sintáticas.

PRODUÇÃO	REGRAS SEMÂNTICAS
$S \rightarrow E$	$E.h = g(E.s)$ $S.s = E.s$
$E \rightarrow E_1 F$	$E_1.h = f_1(E.h, E_2.s)$ $E_2.h = f_2(E.h, E_1.s)$ $E.s = f_3(E_1.s, E_2.s)$
$E \rightarrow \mathbf{id}$	$E.s = \mathbf{id}.v$ $E.s = h(E.h, v)$

Fig. 5.57. Atributos sintetizados e herdados que podem ser avaliados juntos.

o atributo $E.s$ não depende de quaisquer atributos herdados, a função f_3 são passados parâmetros correspondentes a valores de atributos

Definições Dirigidas pela Sintaxe Fortemente Não-Circulares

Os avaliadores vistos nesta seção podem ser construídos para qualquer classe de definições dirigidas pela sintaxe, chamadas de "fortemente não-circulares". Para uma definição desta classe, os atributos associados com um símbolo não terminal podem ser avaliados de acordo com a mesma ordem parcial. Quando for construída uma função para um atributo sintetizado de um não terminal, esta ordem é usada para selecionar os atributos sintetizados que vão ser os parâmetros da função.

Daremos agora uma definição desta classe e mostraremos que esta definição dirigida pela sintaxe na Fig. 5.57 não nesta classe. Fornecemos, então, um algoritmo para testar a circularidade e a não-circularidade forte e mostraremos como implementação do Exemplo 5.30 se estende a todas as definições fortemente não-circulares.

Cada nó numa árvore sintática A tem associado a ele um grafo de dependências. O grafo de dependências para a árvore gramatical pode em geral ter percursos que conectam um atributo de um nó a outro passando para cima e para baixo nos atributos dos outros nós na árvore gramatical e terminando num dado atributo de A. Para nossos propósitos é suficiente pesquisar por percursos que se originam e terminam dentro da árvore gramatical abaixo de A. Um ponto de raciocínio envolve tais percursos partindo de algum atributo herdado de A para algum outro atributo sintetizado de A. Iremos fazer uma estimativa excessivamente pessimista do conjunto de tais percursos através da consideração de ordens parciais sobre os atributos de A.

Seja a produção $A \rightarrow X_1 X_2 \ldots X_n$, $n \geq 1$, com A no lado direito. Seja R uma ordem parcial sobre os atributos de A, para $1 \leq i \leq n$, escrevemos $D_i[R]$ para o grafo obtido pela adição de ladas a D_i como segue: se R ordena o atributo $A.b$ antes de $A.c$, adicionamos um lado de $A.b$ para $A.c$.

Uma definição dirigida pela sintaxe é dita ser *fortemente não-circular* se para cada não terminal A podemos encontrar uma ordem parcial RA sobre os atributos de A, tal que, para cada produção $A \rightarrow X_1 X_2 \ldots X_n$ com lado esquerdo A e não terminais $X_1, X_2, \ldots, X_n$ ocorrendo no lado direito:

1. $D_1[R_1, R_2, \ldots, R_n]$ for acíclico; e
2. se existir um lado do atributo $A.b$ para $A.c$ em $D_1[R_1, R_2, \ldots, R_n]$ então R_A ordena $A.b$ antes de $A.c$.

Exemplo 5.31. Seja p a produção $E \rightarrow E_1 F$ da Fig. 5.57, com grafo de dependências D_p exibido no centro da Fig. 5.58. Seja RE a relação de ordem parcial mínima neste exemplo $E \rightarrow E_1 F$ existem duas ocorrências de não-terminais no lado direito de regras, E_1 e F, como de praxe. Por conseguinte, RE é RE são os mesmos que RF e o grafo $D_1[RE, RF]$ é o grafo mostrado na Fig. 5.61.

Dentre os atributos associados à raiz E da Fig. 5.61, os únicos percursos são de s para s. Como RE faz preceder s não existe violação da condição 2). □

Dada uma definição fortemente não-circular e uma ordem parcial RA para cada não-terminal A, a função f_i é o i-ésimo atributo de A com os argumentos como segue: se RA ordena o atributo herdado a antes de b, então a é um argumento para a função cuja contagem não é.

Um Teste de Circularidade

Uma definição dirigida pela sintaxe é dita circular se o grafo de dependências para algum árvore gramatical possui um ciclo. As definições circulares são malformadas e sem significado. Não existe forma que possamos começar a computar quaisquer valores de atributos no ciclo. Uma coleção de relações de ordem parcial tal que assegurem que uma definição seja fortemente não-circular está estritamente relacionada ao teste de circularidade de uma definição. Iremos, por conseguinte, considerar primeiro um teste para a circularidade

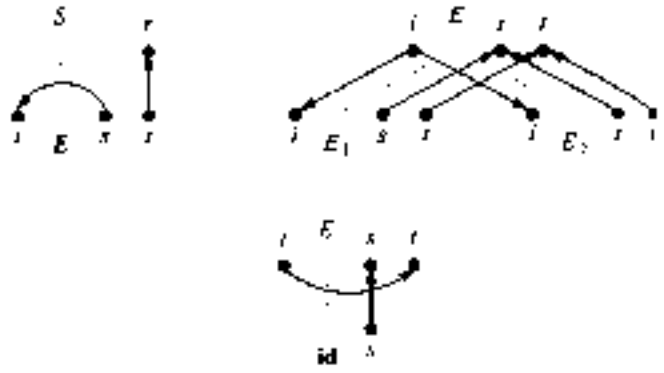


Fig. 5.58. Grafos de dependências para as produções da Fig. 5.57.

```
função L_Atributado(r: nó_da_árvore_sintática);
    var c: nó_da_árvore_sintática;
        k: tipo_do_valor_do_atributo;
    procedimento R;
    início
        se filho_direito = op_aditivo então início
            herdado_op_aditivo := herdado;
            recursão_direita(op_aditivo);
            c := criar_nó(herdado_op_aditivo, c, k)
        fim
    fim;
início
    r := T; R
    retornar c
fim;
```

Fig. 5.56. Compilador preditivo com ações incorporadas para o esquema na Fig. 5.55.

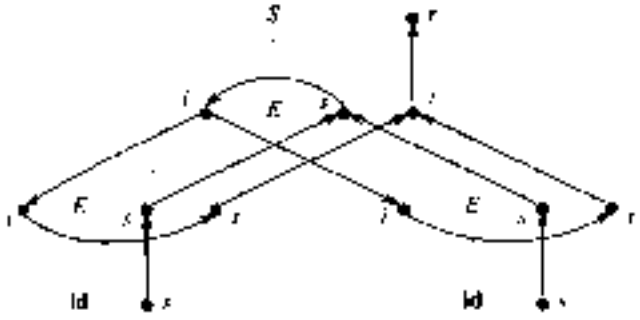


Fig. 5.59. Grafo de dependências para uma árvore gramatical.

Exemplo 5.32. No seguinte definição dirigida pela sintaxe os percursos entre os atributos de Y dependem de que produção é aplicada. Se $A \to \ldots$ for aplicada, então $Y.s$ depende de $Y.i$; em caso contrário, não depende. Para uma completa indeterminação sobre as possíveis dependências teremos que calcular os conjuntos de relações de ordem parcial sobre os atributos de um nó não-terminal.

PRODUÇÃO	REGRAS SEMÂNTICAS
$S \to A$	$A.i = 1$
$A \to Y$	$A.s = Y.s$
$A \to Y$	$A.s = Y.i$

```
Função Ler-Gdp;
    Início
        caso a produção ao nível seja
        'S → A'
            G1 := Ler-Gdp(Nível+1);
            G2 := Ler-Gdp(Nível+1);
            retornar Juntar(G1, G2)
        'A → id'
            retornar Id;
        default:
            ...
        fim
    fim;

    Função Juntar(G1, G2);
    Início
        caso a produção ao nível seja
        'A → B C'
            G := Juntar(G1, G2);
            G := Juntar(Grafo-Dep(Nível+1), G);
            G := Juntar(Grafo-Dep(Nível+2), G);
            retornar Juntar(G, G2);
        'A → id'
            retornar Id;
        default:
            ...
        fim
    fim;

    Função Ativos;
    início
        G := Ler-Grafo(Nível+1);
        G := Ativos(G);
        G := Escrever-Grafo(Nível, G);
        retornar G
    fim;
```

Fig. 5.60. Funções com atributos sintetizados para a Fig. 5.57.

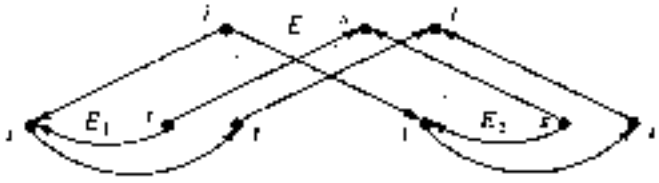


Fig. 5.61. (a, b) conjuntos de dependências para uma produção.

A rotina principal do Algoritmo da Fig. 5.62 é como se segue. Representamos as relações de ordem parcial por grafos dirigidos acíclicos. Dados dois GDAs para os atributos dos símbolos à direita de uma produção, podemos determinar um GDA para os atributos do lado esquerdo como se segue.

Seja a produção p igual a $A \to X_1 X_2 \ldots X_n$ com grafo de dependências D. Seja D_i um GDA para X_i, $1 \leq i \leq n$. Cada lado X_i em D é temporariamente adicionado ao grafo de dependências de D_i para a produção. Se o grafo resultante possui um ciclo, a definição dirigida pela sintaxe é circular. Em caso contrário, os percursos no grafo resultante determinam um novo GDA para os atributos no lado esquerdo da produção. O GDA resultante é adicionado a $A(x)$.

O teste de circularidade da Fig. 5.62 gasta um tempo que é uma exponencial do número de grafos nos conjuntos $A(x)$. Para qualquer árvore gramatical X e sistema definições dirigidas pela sintaxe que não podem ser testadas pela circularidade não é tempo polinomial.

Podemos converter o Algoritmo da Fig. 5.62 num teste mais eficiente, se uma definição cumprir certas hipóteses razoáveis. Calculamos como se segue. Antes de manter uma família de grafos $A(X)$ para cada X, sumarizamos a dependência. Constrói-se um único grafo $DAG(X)$. Note-se que cada grafo em $A(X)$ possui os mesmos nós para os atributos de X, mas pode ter todos diferentes. $DAG(X)$ é o grafo sobre esses nós para os atributos de X que tenha um lado entre $X.b$ e $X.c$ se algum grafo em $A(X)$ tiver. $DAG(X)$ representa uma "estimativa do pior caso" das dependências entre os atributos de X. Em particular, se $DAG(X)$ for acíclico, a definição dirigida pela sintaxe é garantida como não-circular. No entanto, a recíproca não é verdadeira necessariamente, isto

```
para o símbolo gramatical X faça
    DAG provisório inicial com os atributos de X e nenhum
    lado;
repetir
    modificar := falso;
  para a produção p dada por A → X1 ... Xn faça início
    para GDAs G1, ..., Gn para X1, ..., Xn faça início
        D := Dp;
        para cada lado b → c em G1 ∪ ... ∪ Gn faça
            adicionar um lado em D entre os atributos b e c de X
        se D possui um ciclo então
            falhar o teste de circularidade
        senão início
            G := um novo grafo com nós para os atributos
                 de A e nenhum lado;
            para cada par de atributos b e c de A faça se existe
            um percurso em D de b para c então
                    adicionar b → c a G;
            se G não existir ainda em A(A) então início
                    adicionar G a A(A);
                    modificar := verdadeiro
            fim
        fim
    fim
  fim
até que modificar = falso
```

Fig. 5.62. Um teste de circularidade.

E, se *D* possui um ciclo, não é necessariamente verdadeiro que a definição dirigida pela sintaxe seja circular.

O teste de circularidade modificado constrói grafos acíclicos *D*(*X*) para cada *X* se o teste for bem-sucedido. A partir desses grafos podemos construir um avaliador para a definição dirigida pela sintaxe. O método é uma generalização direta do Exemplo 5.30. A função para o atributo sintetizado *X*.s recebe como argumentos todos e somente os atributos herdados que precedem s em *D*(*X*). A função, chamada ao nó n, chama outras funções para computar os atributos sintetizados necessitados ao filho de n. Para as rotinas que computam esses atributos são passados os valores para os atributos herdados dos quais necessitam. O fato de que um teste de não-circularidade tenha tido sucesso garante que esses atributos herdados podem ser computados.

EXERCÍCIOS

5.1 Para a expressão de entrada [illegible], construa uma árvore gramatical anotada de acordo com a definição dirigida pela sintaxe da Fig. 5.2.

5.2 Construa as árvores gramaticais e sintáticas para a expressão [illegible] de acordo com

a) A definição dirigida pela sintaxe da Fig. 5.9 e

b) O esquema de tradução da Fig. 5.28.

5.3 Construa um GDA e identifique os valores numéricos para as subexpressões da expressão seguinte, assumindo que + associe a partir da esquerda.

[illegible]

***5.4** Dê uma definição dirigida pela sintaxe para traduzir expressões infixas em expressões infixas sem parênteses redundantes. Por exemplo, uma vez que + e * associam à esquerda, [illegible] pode ser reescrita como [illegible].

5.5 Dê uma definição dirigida pela sintaxe que gere a derivada de expressões formadas pela aplicação dos operadores aritméticos + e * à variável *x* e a constantes; por exemplo, [illegible]. Assuma que nenhuma simplificação tome lugar, de tal forma que [illegible] é avaliado em [illegible].

5.6 A seguinte gramática gera expressões formadas pela aplicação do operador aritmético + a constantes inteiras e reais. Quando dois inteiros são adicionados, o tipo resultante é inteiro, em caso contrário é real.

$$E \to E + T \mid T$$
$$T \to \textbf{num . num} \mid \textbf{num}$$

a) Dê uma definição dirigida pela sintaxe para determinar o tipo de cada subexpressão.

b) Estenda a definição dirigida pela sintaxe de (a) para traduzir na notação posfixa bem como determinar os tipos. Use o operador unário **inttoreal** para converter um valor inteiro num valor real equivalente, de forma que ambos os operandos de + na forma posfixa tenham o mesmo tipo.

5.7 Estenda a definição dirigida pela sintaxe da Fig. 5.22 de forma a controlar o comprimento dos quadros em adição ao controle de suas larguras. Assuma que o terminal **texto** possua o atributo sintetizado *h* que forneça o comprimento normalizado do texto.

5.8 Seja o atributo sintetizado *val* que forneça o valor do número binário gerado por *S* na gramática seguinte. Por exemplo, para a entrada [illegible], *S.val* = 5.625.

$$S \to L . L \mid L$$
$$L \to L B \mid B$$
$$B \to 0 \mid 1$$

a) Use atributos sintetizados para determinar *S.val*.

b) Determine *S.val* com uma definição dirigida pela sintaxe na qual o único atributo sintetizado de *B* é *c*, fornecendo a contribuição do bit gerado por *B* para o valor final. Por exemplo, as contribuições dos primeiro e último bits em [illegible] para o valor 5.625 são, respectivamente, 4 e 0.125.

5.9 Reescreva a gramática subjacente na definição dirigida pela sintaxe do Exemplo 5.3 de tal forma que a informação de tipo possa ser propagada usando-se somente atributos sintetizados.

***5.10** Quando os enunciados gerados pela gramática seguinte forem traduzidos em um código de máquina abstrata, um enunciado break é traduzido como um desvio para a instrução seguinte ao enunciado while envolvente mais próximo. Por simplicidade, as expressões são representadas pelo terminal **expr** e outros tipos de enunciados pelo terminal **other**. Esses terminais possuem um atributo sintetizado *code* fornecendo suas traduções.

$$S \to \textbf{while expr do begin } S \textbf{ end}$$
$$\mid S ; S$$
$$\mid \textbf{break}$$
$$\mid \textbf{other}$$

Forneça uma definição dirigida pela sintaxe que traduza enunciados em código para a máquina de pilha da Seção 2.8. Assegure-se de que os enunciados break dentro de enunciados while aninhados sejam traduzidos corretamente.

5.11 Elimine a recursão à esquerda das definições dirigidas pela sintaxe nos Exercícios 5.6(a) e (b).

5.12 As expressões geradas pela gramática seguinte podem ter atribuições dentro de si.

$$S \to E$$
$$E \to E := E \mid E + E \mid (E) \mid \textbf{id}$$

A semântica das expressões é como em C. Isto é, [illegible] é uma expressão que atribui o valor de [illegible]; o r-valor desta expressão é o mesmo de [illegible]. Sobretudo, [illegible] atribui o valor de [illegible] e então a [illegible].

a) Construa uma definição dirigida pela sintaxe para verificar se o lado esquerdo de uma expressão é um l-valor. Use o atributo herdado *lado* para o não-terminal *E* a fim de indicar se a expressão gerada por *E* aparece do lado esquerdo ou direito de uma atribuição.

b) Estenda a definição dirigida pela sintaxe em (a) para gerar código intermediário para a máquina de pilha da Seção 2.8 à medida que examina a entrada.

5.13 Reescreva a gramática subjacente do Exercício 5.12 de tal forma que agrupe as subexpressões de := à direita e as subexpressões de + à esquerda.

a) Construa um esquema de tradução que simule uma definição dirigida pela sintaxe do Exercício 5.12(b).

b) Modifique o esquema de tradução de (a) de forma que emita o código incrementalmente para um arquivo de saída.

5.14 Forneça um esquema de tradução que verifique se o mesmo identificador não aparece duas vezes numa lista de identificadores.

5.15 Suponha que as declarações sejam geradas pela seguinte gramática

$$D \to \textbf{id } L$$
$$L \to , \textbf{id } L \mid : T$$
$$T \to \textbf{integer} \mid \textbf{real}$$

a) Construa um esquema de tradução para inserir o tipo de cada identificador na tabela de símbolos, como no Exemplo 5.3.

b) Construa um tradutor preditivo a partir do esquema de tradução gerado.

5.16 A gramática gerando uma versão ambígua da gramática subjacente da Fig. 5.22. As regras [1] são usadas somente para resolver quaisquer ações ambíguas durante a análise.

$$
\begin{aligned}
N &\to E \\
E &\to E\ R\ E \\
R &\to E \text{ sub } E \mid \\
A &\to \mid \ldots \mid \text{texto}
\end{aligned}
$$

a) Adapte a definição dirigida pela sintaxe da Fig. 5.22 de forma que use a gramática acima.

b) Converta a definição dirigida pela sintaxe de (a) em um esquema de tradução.

5.17 Escolha uma transformação para eliminar a recursão na Seção 5.5 e forneça o esquema para o não-terminal Y em 5.2 e

a) Atributos herdados através de regras de cópia

b) Atributos herdados

5.18 Elimine a recursão à esquerda do esquema de tradução do Exercício 5.7(a).

*5.19 Suponhamos termos definição L-atribuída cuja gramática subjacente é LL(1) e que queiramos para as quais podemos resolver ambiguidades e construir uma análise sintática preditiva. Mostre que pode-se armazenar os atributos herdados e sintetizados na pilha da analisador sintático *empregando* as regras dadas pela tabela sintática preditiva.

*5.20 Prove que a adição de não-terminais marcadores finais a uma gramática LL(1) resulta numa gramática que é LR(1).

5.21 Considere a seguinte modificação da gramática LR(1) $(\ldots)$ $\ldots$:

$$
\begin{aligned}
L &\to M\ L\ N \mid \ldots \\
M &\to \epsilon
\end{aligned}
$$

a) Em que sentido esta gramática modificada não é LR(1)? Aplique as regras gramatical para a criação de estados $\ldots$

b) Mostre que a gramática modificada não é LR(1).

*5.22 Mostre que num esquema de tradução baseado em Fig. 5.30, o valor do atributo herdado $N.h$ está sempre imediatamente abaixo do lado direito sempre que $reduz(\ldots)$ não lado direito para B.

5.23 (a) Algoritmo 5.3 para tradução e análise sintática *bottom-up* com atributos herdados indexados não-terminais marcadores para guardar valores dos atributos herdados em posições previstas na pilha do analisador sintático. Pontos onde atributos podem ser avaliados e os valores locais colocados numa pilha separada da pilha sintática.

a) Converta a definição dirigida pela sintaxe na Fig. 5.30 num esquema de tradução.

b) Modifique o esquema de tradução construído em (a) de tal forma que o valor do atributo herdado p seja armazenado na pilha separada. Elimine o não-terminal marcador M no processo.

*5.24 Considere a tradução durante a análise sintática descrita nos Exercícios 5.23 (a). C. Johnson sugere o seguinte método para simular uma pilha separada para os atributos herdados, usando uma variável global para cada atributo herdado. No seguinte produção, o valor é empilhado na pilha p pela primeira ação e removido pela segunda:

$$
A \to \alpha\ \{empilhar(p, e_1)\}\ \beta\ \{desempilhar(p)\}
$$

A pilha p pode ser manipulada pelas seguintes produções que usam uma variável global g e um não-terminal marcador M com o atributo sintetizado s:

$$
\begin{aligned}
A &\to \ldots M[g]\ \beta\ \{p := M.s\} \\
M &\to \epsilon\ \{M.s := g; g := e_1\}
\end{aligned}
$$

a) Aplique essa transformação ao esquema de tradução do Exercício 5.23(b). Substitua todas as referências ao topo da pilha separada por referências à variável global.

b) Mostre que o esquema de tradução construído em (a) computa os mesmos valores para o atributo sintetizado como a solução de pilha que o do Exercício 5.23(b).

5.25 Use a abordagem da Seção 5.6 para implementar todos os atributos de *Khazdr* usando o esquema de tradução do Exercício 5.12(b), através de uma única variável *bordeaux*.

5.26 Modifique o uso da pilha durante uma travessia em profundidade (Exemplo 5.26) de tal forma que os valores na pilha correspondam aos nós na pilha da análise sintática do Exemplo 5.10.

NOTAS BIBLIOGRÁFICAS

O uso de atributos sintetizados para especificar a tradução de uma linguagem apareceu em Irons [1961]. A ideia de uma análise sintática guiada por tradução sintática é discutida em Sandewall e Saale [1969], e Brandon e Morris [1982]. Posteriormente esses atributos herdados, os graus de dependências e um teste pela não-circularidade foram apresentados em Knuth [1968], um teste de circularidade apareceu num artigo de correção. Um exemplo expandido no artigo original. Outras observações dos atributos globais são feitas através de uma gramática geral. Se os atributos podem ser funções, os atributos herdados podem ser eliminados como feito na semântica denotacional, pedindo a associação uma função dos atributos herdados para os sintetizados com a não-terminal. Tais observações aparecem em Mayoh [1981].

Uma aplicação na qual os efeitos colaterais nas regras semânticas são indesejáveis está na edição dirigida pela sintaxe. Suponhamos que um editor seja gerado a partir de uma gramática de atributos para a linguagem fonte, como em Reps [1984], e consideremos as operações de edição no programa-fonte que resulte na remoção de uma parte da árvore gramatical para a programa. Na medida em que não haja efeitos colaterais, as funções do Polo-Perto programa modificado podem ser recomputadas incrementalmente.

Einhoven [1958] usa *markings* para controlar as sobrepressões em arquivos.

A definição de gramáticas L-atribuídas em Lewis, Rosenkrantz e Stearns [1974] é motivada pela tradução durante a análise sintática. Restrições similares sobre as dependências entre os atributos se aplicam a uma única das travessias em profundidade, da esquerda para a direita, em Bochmann [1976]. Gramáticas afixas, como introduzidas por Koster [1971], estão relacionadas às gramáticas L-atribuídas. As restrições sobre as gramáticas L-atribuídas são propostas por Kennedy e Ralha [1980] para controlar o acesso aos atributos globais.

A construção mecânica de avaliadores preditivos, similar aqueles construídos pelo Algoritmo 5.2 é descrita por Bochmann e Ward [1978]. A impressão de que a análise sintática *top-down* permite maior flexibilidade para a tradução mostra-se falsa para uma prova em Brosgol [1974] de que o esquema baseado numa gramática LL(1) pode ser avaliado durante a análise sintática LR(1). Independentemente, Watt [1977] usou não-terminais marcadores para assegurar que os valores dos atributos herdados apareçam na pilha durante a análise sintática *bottom-up*. As posições nos lados direitos das produções onde os não-terminais marcadores podem ser inseridos sem segurança sem se perder a propriedade LR(1) são estudadas em Purdom e Brown [1980] e no Exercício 5.21. Simplesmente exigir que os atributos herdados sejam definidos através de regras de cópia não é bastante para assegurar que os atributos possam ser avaliados durante a análise sintática *bottom-up*; as condições de suficiência sobre as regras semânticas são dadas por Jones e Madsen [1980]. Como um exemplo da tarefa de tradução

que não possa ser tratado de maneira satisfatória. Giegerich e Wilhelm [1978] consideram a geração de código para expressões *boolean*. Veremos na Seção 8.6 que a otimização de *expressões boolean* pode ser usada para esse problema e consequentemente uma segunda passagem completa não é necessária.

Um número de ferramentas especializadas para a implementação de estratégias dirigidas pela sintaxe foram desenvolvidas, começando com TOOL/80, por Fang [1972], mas poucas têm sido vistas em publicações. PQCC, por Leverett [1977], constrói um grafo de dependências em tempo de compilação. O espaço é economizado através do controle dos tempos de vida dos atributos e eliminação das regras de cópia. Métodos de avaluação de atributos baseados em árvores sintáticas são discutidos por Kennedy e Ramanathan [1979] e Odnert e Harry [1979].

Métodos de avaliação de atributos são examinados de forma extensiva e rigorosa por Engelfriet [1984]. Um artigo correlato por Courcelle [1984] pesquisa os fundamentos teóricos. HLP, descrito por Räihä et al. [1983], realiza avaliações alternativas com profundidade, como sugerido por Jazayeri e Walter [1975], e INCOMIST por Farrow [1984] também realiza passagens alternativas. Ganzinger et al. [1982] reportam que MUG permite que a ordem na qual os filhos de um nó são visitados seja determinada pela produção atual. GAG, descrito por Kastens, Hutt e Zimmermann [1982], permite repetidas visitas aos filhos de um nó. GAG implementa a classe primitiva de atributos ordenados definida por Kastens [1980]. A idéia de repetidas visitas aparece num artigo anterior por Kennedy e Warren [1976], onde avaliadores de uma classe maior de gramáticas foram gerados. Os valores armazenados são restringidos a inteiros. Räihä [1978] descreve uma modificação do método de Kennedy e Warren que reserva espaço somente para os valores de atributos numa passagem se forem necessários mais do que uma vez posteriormente. Uma implementação deste método por Brandt et al. [1981] considera vários esquemas para esta classe. Avaliadores recursivos também são construídos por Katayama [1984]. Um enfoque mais geral é discutido em NEATS, por Madsen [1980], onde um GDA é construído para expressões que representam valores de atributos.

A avaliação dependente em tempo de construção do compilador pode economizar tempo e espaço em tempo de compilação. O teste de circularidade é um problema típico de análise. Jazayeri, Ogden e Rounds [1975] provam que um teste de circularidade exponencial no tempo é necessário para gramáticas arbitrárias. Técnicas para aproximar a implementação dos testes de circularidade são consideradas por Ludwig e Raiha [1975], Raiha e Saarinen [1982] e Deransart, Jourdan e Lorho [1984].

Otimizações feitas por avaliadores ingênuos levaram ao desenvolvimento de técnicas para preservação de espaço. Organizado por alocação de valores de atributos a registradores, na Seção 8.8 descrevemos um teste diferente por Mayoh [1981]. O problema de se encontrar uma classificação especial do grafo de dependências que minimize o número de registradores usados é NP-completo em Sethi [1975]. A utilização mínima de compilação de tempo é discutida em avaliador de múltiplas passagens apresentado em Rajlich [1981] e Jazayeri e Pozefsky [1981]. Bochmann et al. [1976] mencionam o uso de pilhas separadas para guardar atributos selecionados e herdados durante uma passagem. O GAG realiza a análise dos tempos de vida dos valores dos atributos e variáveis globais, pilhas e arquivos gerenciados, resultados nas versões. Uma comparação das técnicas de conservação de espaço usadas por GAG e INCOMIST é feita por Farrow e Yellin [1987].

Capítulo 6

VERIFICAÇÃO DE TIPOS

Um compilador precisa verificar se o programa segue as convenções sintáticas e semânticas da linguagem fonte. Essa checagem chama-se *verificação estática* para distingui-la da verificação *dinâmica* durante a execução do programa-objeto. Assegura que certos tipos de erro de programação serão detectados e reportados. Os exemplos de verificação estática incluem:

1. *Consistência de tipos.* Um compilador deveria relatar um erro se um operador for aplicado a um operando inexpressivo. Por exemplo, se um vetor for somado a um apontador, ou se uma variável função for passada como parâmetro.

2. *Verificação de fluxo de controle.* Os comandos que fazem o fluxo de controle devem causar uma construção pretendida. Algum local para onde transferir o controle. Por exemplo, uma ocorrência *break* em C ou comando *exit* deve ter *loop* ou *switch* envolvente; a ocorrência interna com o mesmo nome se um tal comando envolvente existir.

3. *Unicidade de identificadores.* Existem situações nas quais um objeto precisa ser definido exatamente uma vez. Por exemplo, em Pascal um identificador precisa ser declarado unicamente dentro do escopo em que é definido e não pode ser definido mais de uma vez. Rótulos com o mesmo nome precisam ser distintos, e os elementos numerados de uma lista não podem ser duplicados.

4. *Verificações relacionadas aos nomes.* Algumas vezes, o mesmo nome precisa figurar duas ou mais vezes. Por exemplo, em Ada um laço pode ser identificado com um nome opcional após o fim da construção. O compilador precisa verificar se o mesmo nome existe em ambos os locais.

Neste capítulo focalizamos a verificação de tipos. Como os exemplos acima indicam, a maioria das outras verificações estáticas é bastante simples de ser implementada, especialmente se for feita durante a geração de código intermediário. Uma vez que a verificação de tipos exige uma análise sintática e geração de código intermediário como pré-requisitos, ela pode ser convenientemente feita em passagem separada de verificação de tipos entre a análise sintática e a geração de código intermediário, como indicado na Fig. 6.1.

A verificação de tipos checa se o tipo de uma construção corresponde exatamente ao tipo esperado no contexto. Por exemplo, o operador aritmético embutido `+` em Pascal exige operandos inteiros e, portanto, durante a verificação de tipos precisa checar se os dois operandos de `+` têm tipo inteiro. Sendo, porém, a verificação de tipos precisa o suficiente para evitar erros, ela pode ser somente aproximada. Por exemplo, a regra de uma linguagem pode dizer que a função aplicada a um número de argumentos e a um tipo específico de argumento pode ser chamada, mas não dizer que uma função definida pelo usuário seja aplicada ao número e tipos corretos de argumentos cada vez que for chamada. Uma especificação simples de tipo aparece na Fig. 6.2. A verificação dos tipos e a operação de um dado tipo específico são discutidas na Seção 6.3.

As informações a respeito dos tipos capturadas pelo verificador de tipos podem ser necessárias para checagem de código e geração de código. Por exemplo, operadores aritméticos como `+` podem ser aplicados a inteiros ou reais, e em alguns casos o tipo pode ser usado para selecionar a instrução de máquina apropriada. Uma abordagem possível é para determinar o tipo de cada subexpressão. Uma subexpressão possível é a soma de duas expressões com diferentes tipos (como "soma de inteiro com real"). A coerção pode ser implementada pela inserção de operador de conversão explícito em um operador para converter um operando para o tipo esperado pelo operador.

Uma noção sutil de sobrecarga é de "polimorfismo". O corpo de uma função polimórfica pode ser executado com argumentos de vários tipos. Um algoritmo de unificação para inferir tipos das funções polimórficas é discutido neste capítulo.

6.1 SISTEMAS DE TIPOS

O projeto de um verificador para uma linguagem é baseado nas informações sobre as construções sintáticas da linguagem, noções de tipos e das regras para atribuição de tipos a construções das linguagens. Os seguintes exemplos provêm de definições Pascal e de notação de referência de C não, respectivamente, exemplos de informações com que um escritor de compiladores pode se preocupar.

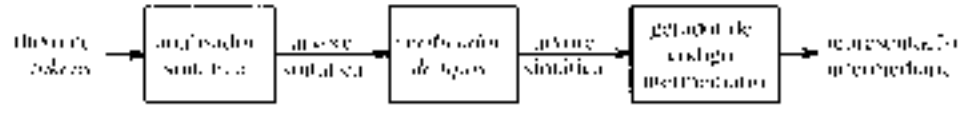


Fig. 6.1. Posição do verificador de tipos

- Se ambos os operandos dos operadores aritméticos de adição, subtração e multiplicação são do tipo inteiro, o resultado também é

- O endereço do operando fornecido & é um apontador para o objeto dado pelo operando. Se o tipo do operando for do tipo , o resultado é um apontador para

Implícito nos exemplos acima é a idéia de que cada expressão possui um tipo associado com ela. Sob estudo, os tipos têm estrutura, e tipos apontador para e construídos a partir de tipos que referência

Tanto em Pascal quanto em C, os tipos ou são básicos ou são construídos. Os tipos básicos são os atômicos, em relação à estrutura interna da unidade em que o programador está envolvido com o sistema. Em Pascal, os tipos básicos são os booleanos, caracteres, inteiros e reais. Tipos subfaixa, como e definidos enumerados, como

podem ser tratados como tipos básicos. Pascal permite que o programador construa tipos a partir dos tipos básicos e de outros tipos construídos, sendo exemplos os arranjos, registros e conjuntos. Adicionalmente, os apontadores e funções podem ser também tratados como tipos construídos.

Expressões de Tipos

O tipo de qualquer construção de linguagem será denotado por uma "expressão de tipo". Informalmente, uma expressão de tipo será um tipo básico ou será formada através da aplicação de um operador, chamado de um *construtor de tipos*, a outras expressões de tipo. Os conjuntos de tipos básicos e de construtores básicos dependem da linguagem que está sendo verificada.

Este capítulo usa a seguinte definição para *expressões de tipo*:

1. Um tipo básico é uma expressão de tipo. Dentre os tipos básicos estão os *booleanos*, *caracteres* e *inteiros*. Um tipo especial de tipo básico, *tipo error*, é usado para sinalizar um erro durante a verificação de tipos. Finalmente, o tipo básico , denotando "a ausência de um valor", permite que todos os comandos sejam verificados.

2. Uma vez que as expressões podem receber nomes, um nome de tipo é uma expressão de tipo. Um exemplo do uso dos nomes de tipo aparece em breve; as expressões de tipo correspondentes são discutidas na Seção 6.3.

3. Um construtor de tipos aplicado a uma expressão de tipo é uma expressão de tipo. Os construtores abrangem:

   a) *Arranjos*. Se T é uma expressão de tipo, então *arranjo*(I, T) é uma expressão de tipo denotando o tipo arranjo de elementos com tipo T e conjunto de índices I. I é frequentemente um intervalo dos inteiros. Por exemplo, a declaração

   associa a expressão de tipo *arranjo*$(1..10, \textit{inteiro})$ a .

   b) *Produtos*. Se T_1 e T_2 são expressões de tipo, então também o são $T_1 \times T_2$. $\times$ é associativo à esquerda.

   c) *Registros*. A diferença entre um registro e um produto é que os campos num registro têm nomes. Um construtor de tipo *registro* será aplicado a uma tupla formada pelos nomes de campos e pelos tipos de campos. Tecnicamente, os nomes de campo deveriam ser parte do construtor de tipos, mas é conveniente manter os nomes de campo junto aos seus tipos associados. No Capítulo 8, o construtor de tipos *registro* é aplicado a um apontador para a tabela de símbolos contendo entradas para os nomes de campo. Por exemplo, o fragmento de programa Pascal

   declara o nome de tipo como representando a seguinte expressão de tipo

   em que denota uma lista de nomes desse tipo.

   d) *Apontadores*. Se T é uma expressão de tipo, então *apontador*(T) é uma expressão de tipo denotando o tipo "apontador para um objeto de tipo T". Por exemplo, em Pascal a declaração

   declara que é um identificador do tipo *apontador*.

   e) *Funções*. Matematicamente, uma função mapeia elementos de um conjunto, o *domínio*, em outro conjunto, o *range*. Devemos tratar as funções nas linguagens de programação como mapeamentos de domínios. Por exemplo, *inteiro* $\rightarrow$ *inteiro* denota um tipo função que mapeia inteiros em inteiros. O tipo de tal função será denotado pela expressão de tipo $D \rightarrow R$. Por exemplo, a função embutida de Pascal possui domínio de tipo e range de tipo , isto é, um par de inteiros, e retorna valor de tipo . Por conseguinte, dizemos que possui o tipo

   Como um outro exemplo, a declaração de função Pascal

   diz que o domínio de tipos de é derivado por *caractere* $\times$ *caractere* e o range é derivado de tipos por *apontador*(*inteiro*). O tipo de é então denotado pela expressão de tipo

   Frequentemente, por motivos de implementação, discutidos no próximo capítulo, existem limitações sobre o tipo que uma função pode retornar; por exemplo, arranjos ou funções não podem ser retornados. Existem, entretanto, linguagens das quais LISP é o exemplo mais proeminente, que permitem retornar objetos de tipos arbitrários e, conseqüentemente, podemos definir uma função do tipo

   Isto é, a função como argumento uma função que mapeia um inteiro em um inteiro e produz como resultado uma outra função do mesmo tipo.

4. Expressões de tipo podem conter variáveis cujos valores sejam expressões de tipo. As variáveis de tipo serão introduzidas na Seção 6.5.

Assumimos que a representação para cada tipo básico e construtor de tipos está disponível. Mais precisamente, assumimos que cada tipo básico e construtor de tipos

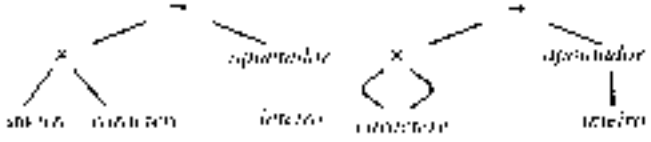


Fig. 6.2. Árvore sintática para uma expressão de tipo — operador de indexação.

Duas questões surgem naturalmente quando discutimos a construção de verificadores de tipos para uma linguagem. Utilizando a abordagem dirigida pela sintaxe apresentada na Seção 5.2, podemos construir uma árvore sintática para uma expressão de tipo e, em seguida, percorrer a árvore para avaliar as regras semânticas para os nós internos que representam construções de tipo e as folhas para nós representando tipos básicos e variáveis de tipo. A Fig. 6.2 ilustra uma árvore desse tipo. Exemplos de representações de expressões de tipo que têm sido usadas em compiladores são fornecidos na Seção 6.3.

Sistemas de Tipos

O sistema de tipos de uma linguagem é uma coleção de regras para inferir expressões de tipo para as várias partes de um programa. Um verificador de tipos implementa um sistema de tipos. Os sistemas de tipos deste capítulo são especificados de uma forma dirigida pela sintaxe, para que possam ser prontamente implementados usando as técnicas do capítulo anterior.

Diferentes sistemas de tipos podem ser usados por diferentes compiladores ou processadores de uma mesma linguagem. Por exemplo, em Pascal, a regra de um array inclui o conjunto de índices do array, portanto, a seguinte função pode ser aplicada a arrays com aquele conjunto de índices. No entanto, muitos compiladores Pascal permitem que o conjunto de índices seja deixado inespecificado quando um array é transmitido como argumento. Por conseguinte, esses compiladores usam um diferente sistema de tipos daquele da definição da linguagem Pascal. Similarmente, o sistema UNIX lint examina programas C procurando possíveis erros, utilizando um sistema de tipos mais detalhado do que o que o próprio compilador C utiliza.

Verificação Estática e Dinâmica de Tipos

A verificação feita por um compilador é dita estática, enquanto que a feita enquanto o programa alvo roda é denominada dinâmica. Em princípio, qualquer verificação pode ser feita dinamicamente, se o código alvo carregar o tipo de cada elemento juntamente com o seu valor.

Um sistema sólido de tipos elimina a necessidade de verificação dinâmica de tipos em busca de erros de tipo, porque nos permite determinar exatamente que esses erros não poderão ocorrer quando o programa alvo rodar. Isto é, se um sistema semântico de tipos atribui um tipo que não corresponde a uma parte do programa, erros de tipo não poderão ocorrer quando o código-alvo para essa parte do programa for posto para rodar. Uma linguagem é fortemente tipada se um compilador puder garantir que o programa que ele aceita será executado sem erros de tipo.

Na prática, algumas verificações só podem ser feitas dinamicamente. Por exemplo, se primeiro declaramos

[illegible]

e, em seguida, computamos [illegible], um compilador não poderá em geral garantir que, durante a execução, o valor de i pertence ao intervalo de 0 a 255.

Recuperação de Erros

Uma vez que a verificação de tipos tem o potencial de capturar erros em programas, é importante para um verificador de tipos realizar algo razoável quando um erro for descoberto. No mínimo, deve ele reportar a natureza e a localização do erro. É desejável que o verificador de tipos se recupere dos erros, de forma a que possa verificar o resto da entrada. Como o tratamento de erros afeta as regras de verificação de tipos, precisa ser projetado no sistema de tipos desde o início; as regras podem ser preparadas para lidar com os erros.

A inclusão do tratamento de erros pode resultar num sistema de tipos que vai além daquele necessitado para especificar programas corretos. Por exemplo, uma vez que o erro tenha ocorrido, podemos não saber o tipo de um fragmento de programa incorretamente formado. Lidar com essa situação requer técnicas similares àquelas discutidas para as linguagens que não requerem que os identificadores sejam declarados antes que sejam usados. As variáveis de tipo, discutidas na Seção 6.6, podem ser usadas para assegurar uma utilização consistente dos identificadores não declarados ou aparentemente incorretamente declarados.

6.2 ESPECIFICAÇÃO DE UM VERIFICADOR SIMPLES DE TIPOS

Nesta seção, especificamos um verificador de tipos para uma linguagem simples, na qual o tipo de cada identificador precisa ser declarado antes de ser usado. O verificador de tipos é uma esquemática de tradução que sintetiza o tipo de cada expressão a partir dos tipos de suas subexpressões. O verificador de tipos pode tratar arrays, apontadores, procedimentos e funções.

Uma Linguagem Simples

A gramática na Fig. 6.3 gera programas, representados pelo não-terminal P, consistindo em uma sequência de declarações D seguidas por uma única expressão E.

Um programa gerado pela gramática da Fig. 6.3 é

[illegible]

Antes de discutir expressões, consideremos os tipos na linguagem. A linguagem tem dois tipos básicos, *boolean* e *integer*, um terceiro tipo básico, *type_error*, é usado para sinalizar erros. Por simplicidade, assumimos que cada array começa por 1. Por exemplo,

[illegible]

leva à expressão de tipo array(1..256, integer). Arrays subsequentemente introduzidos podem ser aplicados ao subintervalo 1..256 da subtipo correto. Como em Pascal, o operador prefixo | nas declarações constrói tipos apontador, e, por conseguinte

[illegible]

$$
\begin{array}{lcl}
P & \rightarrow & D\ ;\ E \\
D & \rightarrow & D\ ;\ D \mid \mathbf{id}\ :\ T \\
T & \rightarrow & \mathbf{char} \mid \mathbf{integer} \mid \mathbf{array}\ [\ \mathit{num}\ ..\ \mathit{num}\ ]\ \mathbf{of}\ T \mid \uparrow T \\
E & \rightarrow & \mathbf{literal} \mid \mathbf{num} \mid \mathbf{id} \mid E\ \mathbf{mod}\ E \mid E\ [\ E\ ] \mid E\ \uparrow
\end{array}
$$

Fig. 6.3. Gramática para uma linguagem simples.

Podemos usar o método de análise de linha descendente, estudado na Seção 5.5, para construir um tradutor para essa linguagem. Em geral, podemos [illegible]

P	→	D ; I	
D	→	D_1 ; D_2	
D	→	**id** : T	{ *coloque tipo de* **id** *em entrada.tipo de* $T.tipo$ }
T	→	**char**	{ $T.tipo$ = *caractere* }
T	→	**integer**	{ $T.tipo$ = *inteiro* }
T	→	↑ T_1	{ $T.tipo$ = *apontador de* $T_1.tipo$ }
T	→	**array** [**num**] **of** T_1	{ $T.tipo$ = *vetor* (**num**.val, $T_1.tipo$) }

Fig. 6.4. Acrescentando tipos apontador e vetor à gramática que reconhece listas de tipos básicos

levará a uma expressão de tipo *apontador vetor de caractere* consistindo do construtor *apontador* aplicado ao tipo *vetor*.

No esquema de tradução da Fig. 6.4, a ação associada à produção $D \rightarrow$ **id** : T grava, para cada identificador, o tipo resultante em uma entrada da tabela de símbolos. A ação *coloque tipo de* **id** *em entrada.tipo de* $T.tipo$ é aplicada ao nó da subárvore de T gerado, que aponta para a entrada de id na tabela de símbolos, e a uma expressão de tipo representada pelo atributo sintetizado $tipo$, para cada exemplo, T.

Se T gera **char** ou **integer**, então $T.tipo$ é definido como sendo *caractere* ou *inteiro*, respectivamente. O número especificado por um *vetor* é obtido a partir do atributo *val* da folha **num** que dá limite superior do índice. O tipo **num** é construído com a operação *tipo_vetor*, que constrói um tipo a partir de um número e de um tipo de elemento.

Como D aparece antes de I no lado direito de $P \rightarrow D$; I, podemos estar certos de que os tipos de todos os identificadores declarados serão salvos antes da expressão gerada por I ser verificada (veja o Capítulo 5). De fato, modificando-se adequadamente a gramática da Fig. 6.4, podemos implementar os esquemas de tradução desta seção durante a análise sintática. *tipo_vetor* e *tipo_apontador* estão definidos

V	→	**id** := E	{ $V.tipo$ = se *info*($E.tipo$) = $E.tipo$ então *tipo_base*($E.tipo$) senão *tipo_erro* }
V	→	**if** E **then** V_1	{ $V.tipo$ = se $E.tipo$ = *booleano* então $V_1.tipo$ senão *tipo_erro* }
V	→	**while** E **do** V_1	{ $V.tipo$ = se $E.tipo$ = *booleano* então $V_1.tipo$ senão *tipo_erro* }
S	→	S_1 ; S_2	{ $V.tipo$ = se $S_1.tipo$ = *vazio* e $S_2.tipo$ = *vazio* então *vazio* senão *tipo_erro* }

Fig. 6.5. Esquema de tradução para verificação de tipos de comandos

Dentro de uma expressão, o operador apontador positivo ↑ produz o objeto apontado por seu operando. O tipo de E ↑ é o tipo do objeto apontado por E:

E	→	E_1 ↑	{ $E.tipo$ = se $E_1.tipo$ = *apontador* então *tipo_base*($E_1.tipo$) senão *tipo_erro* }

Derivadas para reconhecer a atribuição de variáveis e regras semânticas que permitam tipos e operações adicionais dentro das expressões. Por exemplo, para permitir que identificadores tenham tipos *vetor* ou *apontador*, podemos introduzir a produção $E \rightarrow$ **identificador** na gramática da Fig. 6.3. A introdução dos operadores de comparação como = e dos operadores lógicos, como **and** e lógico, nas produções para E permitirá a construção de expressões do tipo *booleano*.

Verificação de Tipos dos Comandos

Uma vez que as construções de linguagem de mais alto nível, como os comandos, não produzem valores, o tipo básico especial *vazio* pode lhes ser atribuído. Se um erro for detectado dentro de um comando, o tipo atribuído ao comando é *tipo_erro*.

Os comandos de uma linguagem de programação incluem expressões, como a atribuição nos exemplos anteriores, e comandos de controle de fluxo, como **if** e **while**. As sequências de comandos são separadas por ponto-e-vírgula (;). As produções da Fig. 6.5 podem ser combinadas com aquelas da Fig. 6.3 se modificarmos a produção para um programa completo em $P \rightarrow D$; S. Um programa nessa forma consiste em declarações seguidas de comandos, as regras acima para a verificação de expressões são ainda necessárias porque os comandos podem ter expressões dentro de si.

As regras para a verificação de comandos são dadas na Fig. 6.5. A primeira regra verifica se os dois lados de um comando de atribuição têm o mesmo tipo. A segunda e a terceira regras especificam que as expressões nos comandos condicionais e comandos **while** possuem tipo *booleano*. Os erros são propagados pela última regra na Fig. 6.5 porque a sequência de comandos possui tipo *vazio* somente se cada subcomando possuir tipo *vazio*. Nessas regras, uma das finalidades de tipos produz o tipo *tipo_erro*, um verificador de tipos mais elaborado, naturalmente, reportaria a natureza e localização da desigualdade de tipos.

A Verificação de Funções

A aplicação de uma função a um argumento pode ser capturada pela produção

$$E \rightarrow E\ (\ E\ )$$

Neste ponto, a gramática permite que o lado esquerdo de uma atribuição seja uma expressão arbitrária. Obviamente, nem todas as expressões podem ser aceitas como destinos válidos para atribuições. Por exemplo, [ ] pode ser aplicado a uma expressão de tipo vetor, mas não é uma expressão válida à esquerda de uma atribuição.

que é uma expressão é a aplicação de uma expressão a outra. As regras para associar expressões de tipo ao não-terminal T podem ser estendidas pelas seguintes produção e ação para permitir tipos de funções em declarações:

$$T \to T_1 \text{ '}\!\to\!\text{' } T_2 \qquad \{ T.tipo := T_1.tipo \to T_2.tipo \}$$

Os resultados em termos de → seta usada como um construtor de função distinguem-se do mesmo símbolo nesta produção.

A regra para verificar o tipo de uma aplicação de função é

$$E \to E_1 (E_2) \qquad \{ E.tipo := \textbf{se } E_2.tipo = s \textbf{ e } E_1.tipo = s \to t \textbf{ então } t \textbf{ senão } tipo_erro \}$$

Esta regra diz que numa expressão formada pela aplicação de E_1 a E_2, o tipo de E_1 precisa ser uma função $s \to t$ proveniente do tipo s de E_2 para algum tipo intercalado t; o tipo de $E_1(E_2)$ é t.

Muitos temas relacionados à verificação de tipos em presença de funções podem ser discutidos com respeito à sintaxe simples acima. A generalização para funções com mais de um argumento é feita pela construção de um tipo produto consistindo nos argumentos. Note-se que n argumentos de tipo $T_1, \ldots, T_n$ podem ser vistos como um único argumento do tipo $T_1 \times \cdots \times T_n$. Por exemplo, poderíamos escrever

[illegible] (6.1)

para declarar uma função root que toma uma função dos reais para os reais e um real como argumentos e retorna um real. A sintaxe no estilo de Pascal para esta declaração é

[illegible]

A sintaxe em (6.1) separa a declaração do tipo de uma função dos nomes de seus parâmetros.

6.3 EQUIVALÊNCIA DAS EXPRESSÕES DE TIPO

As regras de verificação na última seção assumem a forma: "se duas expressões de tipo forem iguais **então** retornar um certo tipo **senão** retornar *tipo_erro*". É por conseguinte importante se ter uma definição precisa de quando duas expressões de tipo são equivalentes. Ambiguidades potenciais afloram quando são dados nomes às expressões de tipo e os nomes são usados em expressões de tipo subsequentes. O tema chave é se um nome numa expressão de tipo vale por si mesmo ou se é uma abreviação para uma outra expressão de tipo.

Uma vez que existe interação entre a noção de equivalência de tipos e a representação de tipos, vamos falar sobre ambas juntas. Por uma questão de eficiência, os compiladores usam representações permitindo uma rápida determinação da equivalência de tipos. A noção de equivalência de tipos implementada por um compilador específico pode frequentemente ser explicada usando-se os conceitos de equivalência de nomes e equivalência estrutural, discutidas nesta seção. A discussão é em termos de uma representação sob a forma de grafos para as expressões de tipo, a qual dá várias folhas para os tipos básicos e para os nomes de tipos e nós interiores para os construtores de tipos, como na Fig. 6.2. Como veremos, tipos definidos recursivamente levam a ciclos no grafo de tipos se um nome for tratado como uma abreviatura de uma expressão de tipo.

Equivalência Estrutural para as Expressões de Tipo

Na medida em que as expressões de tipo são construídas a partir dos tipos básicos e dos construtores, uma noção natural de equivalência entre duas expressões de tipos é a *equivalência estrutural*; isto é, duas expressões ou são do mesmo tipo básico ou são formadas pela aplicação do mesmo construtor a tipos estruturalmente equivalentes. Isto é, duas expressões de tipos são estruturalmente equivalentes se e somente se forem idênticas. Por exemplo, a expressão de tipo *integer* é equivalente somente a *integer* porque ambas representam o mesmo tipo básico. Similarmente, *apontador(integer)* é equivalente somente a *apontador(integer)* porque os dois são formados através da aplicação do mesmo construtor *apontador* a tipos equivalentes. Se usarmos o método dos números de valor do Algoritmo 5.1, para construir uma representação sob a forma de GDA para expressões de tipo, então expressões idênticas de tipos serão representadas pelo mesmo nó.

Na prática, frequentemente são necessárias modificações da noção de equivalência estrutural de forma a refletir as regras efetivas de verificação de tipos da linguagem-fonte. Por exemplo, quando os arrays são transmitidos como parâmetros, podemos desejar não incluir seus limites como parte do tipo.

O algoritmo para testar a equivalência estrutural na Fig. 6.6 pode ser adaptado para testar noções modificadas de equivalência. O algoritmo assume que os únicos construtores de tipos são os arrays, produtos, apontadores e funções. O algoritmo recursivo testa a equivalência estrutural das expressões de tipo sem checar pelos ciclos, de forma que pode ser aplicado a uma representação de árvore ou GDA. Expressões de tipo idênticas não precisam ser representadas pelo mesmo nó no GDA. A equivalência estrutural dos nós em grafos de tipos com ciclos pode ser testada usando um algoritmo da Seção 6.7.

Os limites de *arrays* s e t em

$$s = array(s_1, s_2)$$
$$t = array(t_1, t_2)$$

são ignorados se o teste para a equivalência de *arrays* nas linhas 4 e 5 da Fig. 6.6 for reformulado para

senão se $s = array(s_1, s_2)$ **e** $t = array(t_1, t_2)$ **então**
retornar $sequiv(s_2, t_2)$

Em certas situações, podemos encontrar uma representação para expressões de tipo que seja significativamente mais compacta do que a notação de grafos de tipos. No próximo exemplo, algumas das informações provenientes das expressões de tipos são codificadas como uma sequência de bits, que podem ser então interpretados como um inteiro. A codificação é tal que inteiros distintos representam expressões de tipo estruturalmente não equivalentes. O teste para a equivalência estrutural pode ser acelerado testando-se primeiro pela inequivalência estrutural através da comparação das representações inteiras dos tipos e então aplicando o algoritmo da Fig. 6.6 somente se os inteiros forem os mesmos.

```
(1)   função sequiv(s, t): booleano;
      início
(2)       se s e t são o mesmo tipo básico então
(3)           retornar verdadeiro
(4)       senão se s = array(s1, s2) e t = array(t1, t2) então
(5)           retornar sequiv(s1, t1) e sequiv(s2, t2)
(6)       senão se s = s1 × s2 e t = t1 × t2 então
(7)           retornar sequiv(s1, t1) e sequiv(s2, t2)
(8)       senão se s = apontador(s1) e t = apontador(t1) então
(9)           retornar sequiv(s1, t1)
(10)      senão se s = s1 → s2 e t = t1 → t2 então
(11)          retornar sequiv(s1, t1) e sequiv(s2, t2)
          senão
(12)          retornar falso
      fim
```

Fig. 6.6. O teste de equivalência estrutural de duas expressões de tipos s e t.

Exemplo 6.1 A codificação das expressões de tipo neste exemplo é proveniente de um compilador C escrito por D. M. Ritchie. É também usado pelo compilador Cdecl, descrito em Johnson [1979].

Considerem as expressões de tipo compostas segundo as construções de tipos para apontadores, funções e vetores apresentadas na seção anterior. Para tipos *função*, a função que lista os argumentos que recebem um objeto de tipo *vetor* de tamanho variável ou de algum tamanho indeterminado de elementos de tipo *T*. Note-se que simplificamos os construtores de tipos para ilustrar a função. Iremos controlar o número de elementos que *vetores* variáveis podem ter armazenados em algum local e dessa forma manter parte do construtor de tipo *vetor*. Similarmente, todos os apontadores do construtor *função* de tipo de resultado de uma função e os tipos dos argumentos das funções serão armazenados em algum local. Por conseguinte, operações com expressões estruturalmente equivalentes desse sistema de tipos poderão ainda facilmente ser efetuadas (veja a Fig. 6.4, quando aplicarmos o sistema de tipos a uma verificação de tipos).

Uma vez que cada um desses construtores é uma operação unária, as expressões de tipo criadas pela aplicação desses construtores aos quais *T* pertence também pertencem a esse conjunto. Um caractere Exemplos de tais expressões de tipos são:

```
vetor de T
função sem argumentos retornando T
apontador para função retornando T
vetores apontadores para funções retornando T
```

Cada uma das expressões acima pode ser representada textualmente por sequências de *bits* (representando um esquema de codificação simples). Como existem apenas três construtores de tipos, podemos usar dois *bits* para codificar um construtor como segue:

Construtor de Tipo	Codificação
apontador	01
vetor	10
função	11

Os tipos básicos também podem ser codificados usando-se quantos *bits* forem necessários. Johnson [1979] propôs que os quatro tipos básicos poderiam ser codificados como:

Tipo Básico	Codificação
booleano	0000
caractere	0001
octeto	0010
real	0011

Expressões textuais de tipos podem ser agora codificadas como sequências de *bits*. Os quatro *bits* mais à direita contêm o código do tipo básico mais à esquerda na expressão de tipo. Os dois *bits* seguintes indicam o construtor aplicado ao tipo básico e os dois seguintes descrevem o construtor aplicado àquele último e assim por diante. Por exemplo:

Expressão de Tipo	Codificação
caractere	0000 0001
vetor de caractere	0001 0001
apontador para vetor de caractere	0101 0001
vetor de apontadores para funções retornando caractere	0110 0101

Veja o Exercício 6.17 para mais detalhes.

Além de economizar espaço, tal representação permite que os construtores que aparecem em qualquer expressão de tipo distinta sejam sempre codificados de *bits* que não podem representar o mesmo tipo básico ou construtores em expressão de tipos diferentes. Note-se, também, que tipos de dados poderão ter a mesma sequência de *bits* uma vez que o tamanho de um *vetor* e os argumentos de uma função não estão representados.

A codificação desse exemplo pode ser estendida de forma a incluir tipos de registros. Vamos tratar cada registro como um tipo básico e codificações uma sequência separada de *bits* para cada tipo de campo no registro. Veja o exemplo de tipos em C e examinado posteriormente no Exemplo 6.4. ∎

Nomes para Expressões de Tipos

Em algumas linguagens podem ser dados nomes aos tipos. Por exemplo, no fragmento de programa Pascal:

```
type
  vetor = array [1..10] of integer;
  ponteiro = ^vetor;
var
  a, b : vetor;
  c : ponteiro;
```

(6.2)

vetor e ponteiro são nomes de tipos. Em Pascal, a pergunta "a e b têm todos os mesmos tipos?" só pode ser respondida, corretamente, se se souber a implementação. O problema emerge porque o Relatório Pascal não declara explicitamente "tipo de dado".

Para modelar esta situação, permitamos que expressões de tipo contenham nomes e também que nomes de tipos ocorram em expressões de tipo. Pode-se provavelmente limitar apenas tipos básicos. Por exemplo, [illegible] uma expressão de tipo *apontador* [illegible] uma expressão de tipo. Por enquanto, suponhamos que não existam definições circulares de expressões de tipo, tais como uma que defina [illegible] sendo um nome de uma expressão de tipo contendo [illegible].

Quando os nomes são permitidos em expressões de tipo, duas noções de equivalência emergem, dependendo de como tratamos os nomes. A *equivalência por nome* trata cada nome de tipo como sendo de um tipo distinto e, dessa forma, duas expressões são equivalentes se, e somente se, os nomes forem idênticos. Sob a outra definição, *estrutural*, os nomes são substituídos pelas expressões de tipo que definem e, dessa forma, duas expressões são estruturalmente equivalentes se representarem duas expressões de tipo estruturalmente equivalentes após a substituição de todos os nomes.

Exemplo 6.2. As expressões de tipo que poderiam ser associadas à variável *a* nas declarações do Exemplo 6.1 são dadas pela seguinte tabela:

Variável	Expressão de Tipo
[illegible]	[illegible]
[illegible]	[illegible]
[illegible]	*apontador* [illegible]
[illegible]	*apontador* [illegible]
[illegible]	*apontador* [illegible]

Sob a equivalência por nome, as variáveis [illegible] têm o mesmo tipo, porque foram associadas às mesmas expressões de tipo. As variáveis [illegible] também têm o mesmo tipo. Note-se, porém, que [illegible] não, muito embora suas expressões de tipo associadas sejam diferentes. Sob a equivalência estrutural, todas as variáveis [illegible] têm o mesmo tipo porque [illegible] um nome para a expressão de tipo *apontador* [illegible]. ∎

Os conceitos de equivalência por nome e estrutural são úteis para se explicar as regras usadas pelas várias linguagens a fim de se associar tipos aos identificadores que ocorrem em declarações.

Exemplo 6.3. As definições emergem em Pascal a partir do fato de que muitas implementações associam um nome explícito de tipo a cada identificador declarado. Se uma declaração contém uma expressão de tipo que não seja um nome, um nome implícito é criado. Um processador simplesmente cria um novo nome cada vez que uma expressão de tipo ocorre em uma declaração de variável.

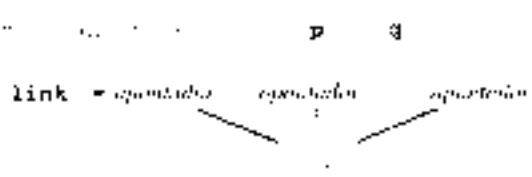


Fig. 6.7. Associação de nomes a nós em um grafo de tipos

Note-se que o nome de tipo link é definido em termos de ponteiro, que por sua vez é definido em termos de link, por conseguinte, suas definições são recursivas.

Porque os tipos recursivos podem ser substituídos se existirem ciclos nos grafos de tipos. Se ponteiro for substituído por a expressão de tipo mostrada na Fig. 6.8(a) é obtida para link. Usando ciclos, como na Fig. 6.8(b), podemos eliminar a referência à parte do grafo de tipos abaixo do nó rotulado registro.

Exemplo 6.4. Ciclos nos grafos de tipos podem ser úteis ao representar tipos de registros. Uma declaração de poderia ser como

```
struct lista {
    int info;
    struct lista *proximo;
} *cabeca, *z;
```

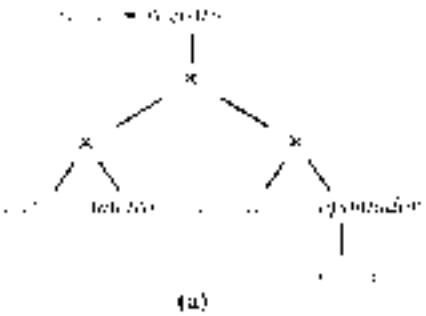
A palavra-chave introduz a declaração de cujo nome se torna parte do tipo do registro. Como exemplo, a representação acíclica da Fig. 6.9(a).

Porque cada um dos nomes de tipos separados de cada um dos nomes de variáveis, exceção feita aos identificadores para tipos registro não declarados — todos os ciclos ponteiros, consequentemente, são derivados apontadores para registros. Uma vez que o nome de um registro é parte de seu tipo, a estrutura cíclica é usada para distinguir entre dois tipos de registro estruturalmente idênticos, tais tipos sendo comparados sucessivamente porque são distintos. Duas declarações de registro com a mesma sintaxe podem, no entanto, ser equivalentes porque são derivadas da mesma declaração.

6-4 CONVERSÕES DE TIPO

Considere-se a expressão $x + i$, onde x é do tipo real e i do tipo inteiro. Como a representação dos inteiros e reais é bastante diferente dentro do computador e diferentes instruções de máquina são usadas para as operações sobre inteiros e reais, o compilador pode ter que primeiro converter um dos operandos de $+$ para assegurar que ambos os operandos sejam do mesmo tipo quando a adição efetivamente for feita.

A determinação da linguagem especifica quando conversões são necessárias. Quando um inteiro é atribuído a um real ou vice-versa, a



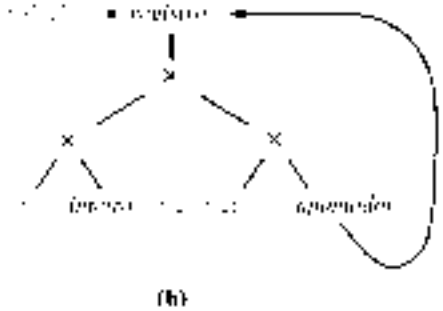


Fig. 6.8. Árvores de tipos com nós compartilhados

conversão deve ser feita para o tipo do lado esquerdo da atribuição. Nas expressões, a transformação usual converte um inteiro para um número real, em seguida, realiza a operação de ponto flutuante resultante de operadores reais. O verificador de tipos também pode ser usado para inserir essas operações de conversão na representação intermediária do programa fonte. Por exemplo, a notação prefixa para `+ 1` poderia ser

```
+ inttoreal 1
```

Aqui o operador `+ inttoreal` converte seu operando para real e em seguida realiza a adição real de seus operandos.

A conversão de tipos frequentemente depende em algum contexto. Um símbolo que tenha diferentes significados dependendo de seu contexto é dito *sobrecarregado*. A sobrecarga será discutida a seguir, porém nesta seção nós nos concentramos apenas nas conversões de tipos que frequentemente acompanham a sobrecarga.

Coerções

A conversão de um tipo para outro é dita *coerção* se for feita automaticamente pelo compilador. Conversões implícitas de tipo, também chamadas de coerções, estão limitadas a certos contextos em muitas linguagens. Em situações onde ocorrem a confusão e a perda de informação são possíveis, por exemplo, um inteiro pode ser convertido para um real mas não vice-versa. Na prática, entretanto, alguma perda de precisão quando um número real precisa caber em uma variável inteira é preferível a nenhuma conversão de tipo, pois frequentemente acompanham a sobrecarga.

A conversão de tipo pode ser proibida, isto é, pode ser necessário algum código explícito para provocar a conversão. Para todos os propósitos práticos, todas as conversões em Ada são explícitas. As conversões explícitas se parecem exatamente com aplicações de funções para um verificador de tipos, e essa é uma opção que será preferida em muitos projetos de linguagens.

Por exemplo, em Pascal, a função embutida na linguagem `ord(c)` retorna o valor inteiro do caractere `c`. Isso realiza o tipo apropriado de conversão de um inteiro para um caractere e vice-versa. Essas conversões são explícitas. Um compilador, porém, também pode converter implicitamente os caracteres ASCII para inteiros na linha entre `'0'` e `'9'` nas expressões aritméticas.

Exemplo 6.5. Consideremos as expressões formadas pela aplicação do operador aritmético `op` a constantes e identificadores, como na gramática da Fig. 6.9. Suponhamos que existam tipos `real` e `integer`, e que inteiros sejam convertidos para reais quando necessário. O atributo `tipo` para o não-terminal `E` pode ser `integer` ou `real`. As regras de verificação de tipos são mostradas na Fig. 6.9. Como na Seção 2.7, a função `ptr(n)` retorna o tipo guardado na entrada da tabela de símbolos apontada por `n`.

PRODUÇÃO	REGRA SEMÂNTICA
`E → num`	`E.tipo := integer`
`E → (num . num)`	`E.tipo := real`
`E → id`	`E.tipo := procurar(id.entrada)`

`E → E1 op E2`	`se E1.tipo = integer`
	 `e E2.tipo = integer`
	 `então integer`
	 `senão se E1.tipo = integer`
	 `e E2.tipo = real`
	 `então real`
	 `senão se E1.tipo = real`
	 `e E2.tipo = integer`
	 `então real`
	 `senão se E1.tipo = real`
	 `e E2.tipo = real`
	 `então real`
	 `senão tipo_erro`

Fig. 6.9. Regra para a verificação de tipos para a coerção de inteiro para real.

A conversão implícita de constantes pode facilmente ser feita em tempo de compilação. Forçar uma coerção em tempo de execução do programa objeto. Nos seguintes fragmentos de código, `i` é uma variável de ponto flutuante e `j` é uma variável inteira. Usando um compilador Pascal, Bentley [1982] encontrou que o fragmento de código

```
i := 0;
while i <> j do i := i + 1
```

é cerca de 4840 vezes mais lento para executar enquanto que o fragmento

```
i := 0.0;
while i <> j do i := i + 1.0
```

leva cerca de 5420. Ambos os fragmentos atribuem valores idênticos a `i` e `j`, porém o código gerado por este compilador para o primeiro fragmento contém uma chamada para uma rotina em tempo de execução que converte a representação interna de uma representação sob a forma de número real. Uma vez que esta rotina em tempo de compilação que `i` é uma variável de reais, um compilador mais completo converteria `1` para `1.0` em tempo de compilação.

6.5 SOBRECARGA DE FUNÇÕES E OPERADORES

Um símbolo *sobrecarregado* é aquele que possui diferentes significados dependendo de seu contexto. Na Matemática, o operador de adição `+` é sobrecarregado, porque `A + B` possui diferentes significados quando `A` e `B` são inteiros reais, números complexos ou matrizes. Em Ada, os parênteses `( )` são sobrecarregados: a expressão `E(3)` pode significar a chamada de função `E` com o argumento `3`, uma chamada para função `E` com argumento `3`, ou uma conversão explícita da expressão `3` para o tipo `E`.

A sobrecarga é resolvida quando um único significado para uma ocorrência de um símbolo sobrecarregado é determinado. Por exemplo, se `+` pode denotar a adição inteira ou real, então as duas ocorrências de `+` em `x + (y + z)` podem denotar diferentes tipos de adição, dependendo dos tipos de `x`, `y` e `z`. A resolução da sobrecarga é algumas vezes mencionada como *desambiguação de operador*, pois determina que operador um símbolo de operação denota.

Os operadores aritméticos são sobrecarregados na maioria das linguagens. No entanto, a sobrecarga envolvendo operadores como `[]` pode ser resolvida olhando-se somente para os argumentos do operador. A análise de caso para se determinar o custo da versão inteira ou real de `+` é similar àquela na regra semântica para `E → E1 op E2` na Fig. 6.9, onde o tipo de `E` é determinado pelos exames dos possíveis tipos de `E1` e `E2`.

O Conjunto de Possíveis Tipos para uma Subexpressão

Nem sempre é possível se resolver a sobrecarga somente através do exame dos argumentos de uma função, como o próximo exemplo mostra. Em lugar de um único tipo, uma subexpressão sobrecarregada pode ter um conjunto de possíveis tipos. Em Ada, o contexto pode prover informações suficientes para reduzir esse conjunto para um único tipo.

Exemplo 6.6. Em Ada, uma das interpretações possíveis de `f(i)` é uma chamada de operador `f` através de uma função de um parâmetro inteiro `i`. O operador pode ser sobrecarregado pela adição de declarações como as seguintes:

PRODUÇÃO	REGRAS SEMÂNTICAS
$E \rightarrow E_1$	$E.tipos = E_1.tipos$
$E \rightarrow \mathbf{id}$	$E.tipos = \{$ *procure* **id**.*entrada*$\}$
$E \rightarrow E_1(E_2)$	$E.tipos = \{ t \mid$ existam t_1 em $E_1.tipos$ e t_2 em $E_2.tipos$ tais que t_1 seja uma função $t_2 \rightarrow t \}$

Fig. 6.10. Determinando o conjunto de possíveis tipos de uma expressão.

Após as declarações acima, os possíveis tipos para *i* incluem

inteiro → *inteiro* — *inteiro*
inteiro → *inteiro* — *complexo*
complexo → *complexo* — *complexo*

Suponhamos que o único tipo possível para E_1 em $E_1(E_2)$ seja inteiro. Com as declarações acima, a subexpressão E_2 pode ter tipo inteiro ou tipo complexo, dependendo do seu contexto. Se a expressão completa for $E_1(E_2)$, então E_2 terá necessariamente o tipo inteiro, porque isso forma um par de inteiros ou um par de números complexos como argumento. Por outro lado, E_2 precisará ter o tipo complexo se a expressão completa for $E_1(E_2)$ em uma declaração como a mostrada.

Na Seção 6.2, assumimos que cada expressão pode ser tipo-verificada pela passagem de atributos sintetizados de tipos para uma aplicação de função como

$E \rightarrow E_1(E_2)$ | $E.tipo = \textbf{se } E_2.tipo = s \textbf{ e } E_1.tipo = s \rightarrow t \textbf{ então } t \textbf{ senão } tipo\_erro$

A generalização natural desta regra para conjuntos de tipos é mostrada na Fig. 6.10. A única operação na Fig. 6.10 é a de aplicação de uma função, as regras para verificar outros operadores com expressões são análogas. Pode haver várias declarações para um operador sobrecarregado, de forma que assumimos que uma entrada numa tabela de símbolos possa ter um conjunto de tipos possíveis, este conjunto é retornado pela função *procure*. O símbolo terminal **id** de posse E gera a expressão completa. Seu papel é exclusivamente sintático.

Expressa em palavras, a terceira regra da Fig. 6.10 diz que se s for um dos tipos de E_2 e um dos tipos de E_1 puder ser $s \rightarrow t$, em que t é um dos tipos de $E_1(E_2)$. Um único tipo de um conjunto durante uma aplicação de função resulta em que o conjunto $E.tipos$ se torne vazio, uma condição que usamos temporariamente para sinalizar um tipo-erro.

Exemplo 6.7. Além de ilustrar a especificação da Fig. 6.10, este exemplo sugere como a abordagem se estende a outras construções. Em particular, consideramos a expressão $i + 1$. Segundo as declarações do operador $+$ como no Exemplo 6.6 (inteiro, inteiro) ou (complexo, complexo), $+$ pode ter tipo inteiro ou tipo complexo, dependendo do contexto. Os conjuntos de possíveis tipos para as subexpressões de $i + 1$ são mostrados na Fig. 6.11, onde i e 1 abreviam *inteiro* e *complexo*, respectivamente.

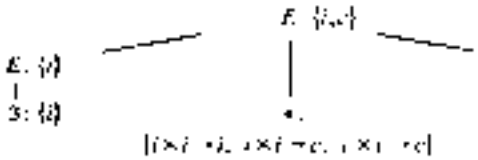


Fig. 6.11. Conjunto de possíveis tipos para as expressões.

De início, suponhamos que o único tipo possível para 1 seja *inteiro*. O operador $+$, portanto, aplica-se a um par de inteiros. Se tratarmos este par de inteiros como uma unidade, seu tipo será dado por *inteiro* → *inteiro*. Existem duas funções no conjunto de tipos para $+$ que se aplicam a pares de inteiros: uma retorna um inteiro e a outra retorna um número complexo. Desde que a raiz pode ter um tipo inteiro ou um tipo complexo.

Estreitando o Conjunto de Possíveis Tipos

Até aqui, que uma expressão completa tenha um único tipo. Dado um único tipo a partir do contexto, podemos estreitar os conjuntos de tipos para cada subexpressão. Se esse processo não resultar num único tipo para cada subexpressão, um erro é declarado para toda a expressão.

Antes de elaborarmos os detalhes para lidar com a sobrecarga, em uma expressão para suas subexpressões, fazemos um exame detalhado nos conjuntos $E.tipos$ construídos pelas regras da Fig. 6.10. Mostramos que cada tipo em $E.tipos$ é um tipo para E e é possível escolher dentre os tipos sobrecarregados dos identificadores em E de uma forma em que E assuma o tipo t. As propriedades vigoram para os identificadores, por declaração, uma vez que cada elemento de **id**.*tipos* é viável. Para o passo indutivo, consideramos o tipo em $E_1(E_2)$, onde $E = E_1(E_2)$. Da regra para a aplicação de função na Fig. 6.10 para algum tipo s, $s \rightarrow t$ estará em $E_1.tipos$ e o tipo s precisará estar em $E_2.tipos$. Por indução, $s \rightarrow t$ é viável para E_1 e s é viável para E_2, respectivamente. Segue que t é um tipo viável para E.

Pode haver várias maneiras de se chegar a um tipo viável. Por exemplo, considerando a expressão $i + 1$, onde $+$ pode ter os tipos $i \rightarrow i \rightarrow i$ e $c \rightarrow c \rightarrow c$, podemos ter os tipos i e c para $i + 1$. Se possuirmos tipos s pode ter os tipos i ou c.

A definição dirigida pela sintaxe da Fig. 6.12 é obtida a partir daquela da Fig. 6.10 pela adição de regras semânticas para determinar um tipo único para E. O atributo sintetizado *unico* de E é discutido abaixo.

Dado que toda a expressão é gerada por E, designamos um E raiz como aquele cujo conjunto $E.tipos$ contém tipos de função cujos tipos de retorno formam o conjunto $E.tipos$. Isso significa que

PRODUÇÃO	REGRAS SEMÂNTICAS
$E \rightarrow E_1$	$E.tipos := E_1.tipos$
	$E.unico := \textbf{se } E.tipos = \{t\} \textbf{ então } t \textbf{ senão } tipo\_erro$
	$E_1.codigo := E.codigo$
$E \rightarrow \mathbf{id}$	$E.tipos := \{$ *procure* **id**.*entrada* $\}$
	$E.codigo := \{$ *gerar* **id**.*lexema* $\}$
	E.unico
$E \rightarrow E_1(E_2)$	$E.tipos := \{ t \mid$ existam t_1 em $E_1.tipos$ e t_2 em $E_2.tipos$ tais que $t_1 = t_2 \rightarrow t$ e t_2 em $E_2.tipos \}$
	$s := E.unico$
	$S := \{ t_2 \mid t_2 \rightarrow s$ em $E_1.tipos \}$
	$E.unico := \textbf{se } S = \{t\} \textbf{ então } t$ $\textbf{senão } tipo\_erro$
	$E_1.unico := \textbf{se } S = \{t\} \textbf{ então } t \rightarrow s$ $\textbf{senão } tipo\_erro$
	$E.codigo := E_1.codigo \; E_2.codigo \;$ $gerar$ '**aplicar**' $E_1.unico$

Fig. 6.12. Especificação do conjunto de tipos de uma expressão.

herdado como o valor de *A* em *a*. Novamente o tipo básico *type_error* assinala o erro.

Se uma função *f* tem a expressão *type x*, podemos encontrar um tipo *t* que seja válido para o argumento *f* ao mesmo tempo em que *t* é válido para a função. O fragmento de programa sintático correspondente na Fig. 6.12 é usado para checar se existe um único tipo *t* com essa propriedade.

A definição dirigida pela sintaxe na Fig. 6.12 pode ser implementada realizando duas travessias em profundidade da árvore sintática para uma expressão. Durante a primeira travessia, o atributo *tipo* é calculado de baixo para cima. Durante a segunda, o atributo *tipo* é propagado de cima para baixo, e na medida em que aplicamos *U* em um nó, o atributo *código* pode ser sintetizado. Na prática, o verificador de tipos pode simplesmente atribuir um único tipo a cada nó da árvore sintática. Na Fig. 6.12, geramos uma notação pós-fixa para sugerir como o código intermediário poderia ser gerado. Na notação pós-fixa, cada identificador e instância do operador **aplicar** possui um tipo anotado e são pela função *apply*.

6.6 FUNÇÕES POLIMÓRFICAS

Um procedimento genérico permite que os conjuntos em seu corpo sejam executados com argumentos de tipos distintos, a cada vez que um procedimento polimórfico é chamado. Os procedimentos em seu corpo podem ser executados com argumentos de diferentes tipos. O termo "polimórfico" pode também ser aplicado a qualquer trecho de código que possa ser executado com argumentos de tipos diferentes e dessa forma podemos falar igualmente de funções e de operadores polimórficos.

Operadores embutidos para indexar arrays e aplicar funções a manipular apontadores são usualmente polimórficos, porque a definição restringe o tipo particular de cada função ou apontador. Por exemplo, a aritmética de referência de C estabelece a respeito do operador de endereço, aplicado a *x* se o tipo do operando for *T*, o tipo do resultado é um apontador para *T*. Uma vez que qualquer tipo pode ser substituído em lugar de *T*, o operador & em C é polimórfico.

Em Ada, as funções "genéricas" são polimórficas, mas o polimorfismo em Ada é restrito. Os termos "genérico" também tem sido usado para referenciar as funções sobrecarregadas e a coerção dos argumentos de funções, termos estes que não discutiremos aqui.

Esta seção endereça os problemas que emergem no projeto de um verificador de tipos para uma linguagem com funções polimórficas. Para ilustrarmos como o polimorfismo expande nossa noção de conjunto de expressões de tipo para incluir expressões com variáveis de tipo. A inclusão de variáveis de tipo levanta algumas questões interessantes relacionadas à equivalência de expressões.

Por que Funções Polimórficas?

As funções polimórficas são frequentes porque facilitam a implementação de algoritmos que manipulam estruturas de dados, independentemente dos tipos de seus elementos. Por exemplo, é conveniente termos um programa que determine o comprimento de uma lista sem termos que saber os tipos dos elementos da lista.

Linguagens como Pascal requerem especificações completas dos tipos dos parâmetros de funções, de forma que uma função para determinar o comprimento de uma lista ligada de inteiros não pode ser aplicada a uma lista de reais. O código Pascal da Fig. 6.13 é para listas de inteiros. A função *length* segue a cadeia de apontadores para o próximo elemento da lista até que um *nil* com valor *nil* seja alcançado. A saída da função não depende de maneira alguma do tipo de informação numa célula. Pascal requer que o tipo de *x* seja declarado quando a função *length(x)* for escrita.

Numa linguagem com funções polimórficas, como ML (Milner [1984]), a função *length* acima pode ser escrita de forma a se aplicar a qualquer tipo de lista, como mostrado na Fig. 6.14. A palavra

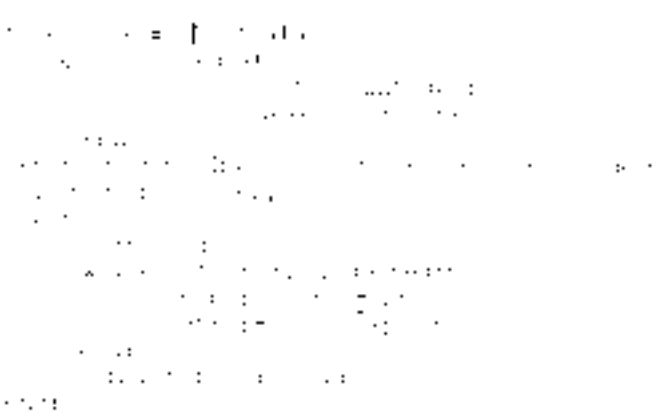


Fig. 6.13. Programa Pascal para o comprimento de uma lista.

chave *fun* indica que *length* é uma função com um argumento *x*. Funções são pré-definidas, e *hd* retorna o cabeça da lista *x* e *tl* retorna o resto da lista após o primeiro elemento ser removido. Com a definição mostrada na Fig. 6.14, as duas seguintes aplicações da função *length* produzem 3:

```
length(["sun","mon","tue"])
length([10,9,8])
```

Na primeira, *length* é aplicada a uma lista de cadeias de caracteres, na segunda, é aplicada a uma lista de inteiros.

Variáveis de Tipo

As variáveis representando expressões de tipo nos permitem falar a respeito de tipos desconhecidos. No restante desta seção, iremos usar letras gregas α, β etc. para variáveis de tipo em expressões de tipo.

Uma aplicação importante das variáveis de tipo é a de verificar uso consistente de identificadores numa linguagem que não requeira que identificadores sejam declarados antes de serem usados. Uma variável representa o tipo de um identificador não declarado. Examinando o programa podemos dizer se os identificadores são usados consistentemente, por exemplo, não encontrando o mesmo identificador em outro lugar. Tal utilização inconsistente pode ser reportada como um erro. Por outro lado, uma variável só sempre usada com outros identificadores consistentemente pode ser aceita como um programa bem formado.

A reconstrução de tipos é o problema de determinar o tipo de uma construção de uma linguagem a partir da forma que foi usada. O termo frequentemente aplicado ao problema de inferir o tipo de uma função a partir de seu corpo.

Exemplo 6.8. As técnicas de inferência de tipos podem ser aplicadas a programas em linguagens como C e Pascal, para prover certas informações de tipos ausentes em tempo de compilação. O fragmento de código na Fig. 6.15 mostra o procedimento *f*, que possui um parâmetro *p* que é ele próprio um procedimento com um parâmetro, mas cujos tipos não são conhecidos. Tais especificações incompletas de tipos são permitidas por C e pela maioria de versões de Pascal.

O procedimento *f* é aplicado a um parâmetro *g*, usado se não uma lista ligada. Por exemplo, *g* pode ser usado para inicializar os inteiros

```
g(a[0]);
g(a[1]);
```

Fig. 6.14. Programa ML para o comprimento de uma lista.

[illegible] A despeito do fato dos tipos dos argumentos [illegible] não serem especificados, podemos inferir, a partir da [illegible] a expressão [illegible] que o tipo [illegible] terá que ser

[illegible]

Qualquer [illegible] com um parâmetro do tipo procedimento [illegible] esse tipo é um erro. Um procedimento pode ser pensado [illegible] função que não retorna um valor e, por conseguinte, o tipo do seu resultado é vazio. □

As técnicas para a inferência e verificação de tipos possuem [illegible] em comum. Em cada caso, temos de lidar com expressões de tipo contendo variáveis. Uma argumentação similar àquela do exemplo a seguir é usada [illegible] nesta seção por um verificador de tipos a fim de inferir os tipos representados pelas variáveis.

Exemplo 6.9. Um tipo pode ser inferido para a função polimórfica [illegible] no seguinte pseudoprograma. A função [illegible] possui o mesmo [illegible] que o operador Pascal ↑ para o desreferenciamento de apontadores.

[illegible]

quando a primeira linha

[illegible]

[illegible] não sabemos nada a respeito do tipo de [illegible] e então vamos representá-lo por uma variável de tipo β. Por definição, o operador [illegible] toma um apontador para um objeto e retorna o objeto. Uma vez que o operador [illegible] é aplicado a uma expressão [illegible] segue-se [illegible] que ser um apontador para um objeto de tipo desconhecido [illegible] então, precisamos aprender que

$$\beta = apontador(\alpha)$$

onde α é uma outra variável de tipo. Sabendo, a expressão [illegible] possui tipo α e, por conseguinte, podemos escrever a expressão de tipo

$$\text{para quaisquer tipos } \alpha, apontador(\alpha) \to \alpha \qquad (6.3)$$

para o tipo da função [illegible] □

Uma Linguagem com Funções Polimórficas

Tudo o que dissemos até então a respeito de funções polimórficas é que podem ser executadas com argumentos de "tipos diferentes". Enunciados precisos sobre o conjunto de tipos a que uma função polimórfica pode ser aplicada são feitos usando-se o símbolo ∀, significando "para qualquer tipo". Consequentemente,

$$\forall\alpha.\ apontador(\alpha) \to \alpha \qquad (6.4)$$

é como escrevemos a expressão de tipo (6.3) para o tipo da função [illegible] no Exemplo 6.9. A função polimórfica [illegible] da Fig. 6.14 toma uma lista de elementos de qualquer tipo e retorna um inteiro e seu tipo pode ser escrito como

$$\forall\alpha.\ lista(\alpha) \to inteiro \qquad (6.5)$$

[illegible]

[illegible]

Fig. 6.15. Procedimento [illegible] com parâmetro tipo procedimento.

Aqui, *lista* é um construtor de tipos. Sem o símbolo ∀, podemos apenas dar exemplos de possíveis domínios e intervalos de tipos para [illegible]

$$lista(inteiro) \to inteiro$$
$$lista(lista(caractere)) \to inteiro$$

As expressões como (6.5) são os enunciados mais gerais que podemos fazer a respeito do tipo de uma função polimórfica.

O símbolo ∀ é o *quantificador universal* e a variável de tipo a qual é aplicado é dita estar *ligada* [illegible] ao mesmo. As variáveis ligadas podem ser renomeadas à vontade, providenciando que todas as ocorrências da variável sejam renomeadas. Por conseguinte, na expressão de tipo

$$\forall\gamma.\ apontador(\gamma) \to \gamma$$

é equivalente a (6.4). Uma expressão de tipo com o símbolo ∀ na mesma será referenciada informalmente como um "tipo polimórfico".

A linguagem que iremos usar para verificar funções polimórficas é gerada pela gramática da Fig. 6.16.

Os programas gerados por esta gramática consistem em uma sequência de declarações seguidas pela expressão E a ser verificada, por exemplo,

$$\forall\alpha.\ apontador(\alpha) \to \alpha\ ; \qquad (6.6)$$
$$apontador(apontador(inteiro))\ ;$$

[illegible]

Minimizamos a notação fazendo com que o não-terminal T gere expressões diretamente. Os construtores → e × formam funções e produtos de tipos. Os construtores unários, representados por **construtor_unário**, permitem que tipos como *apontador(inteiro)* e *lista(inteiro)* sejam escritos. Os parênteses são usados simplesmente para agrupar os tipos. As expressões cujos tipos devem ser verificados possuem uma sintaxe muito simples: podem ser identificadores, sequências de expressões formando uma tupla ou aplicação de uma função a um argumento.

As regras de verificação de tipos para as funções polimórficas diferem de três maneiras daquelas para as funções ordinárias da Seção

$$
\begin{aligned}
P &\to D\ ;\ E \\
D &\to D\ ;\ D \mid \textbf{id} : Q \\
Q &\to \forall\,\textbf{variável_de_tipo}\ .\ Q \mid T \\
T &\to T\ \text{'→'}\ T \\
&\mid T \times T \\
&\mid \textbf{construtor_unário}\ (\ T\ ) \\
&\mid \textbf{tipo_básico} \\
&\mid \textbf{variável_de_tipo} \\
&\mid (\ T\ ) \\
E &\to E\ (\ E\ ) \mid E\ ,\ E \mid \textbf{id}
\end{aligned}
$$

Fig. 6.16. Gramática para linguagem com funções polimórficas.

[illegible]

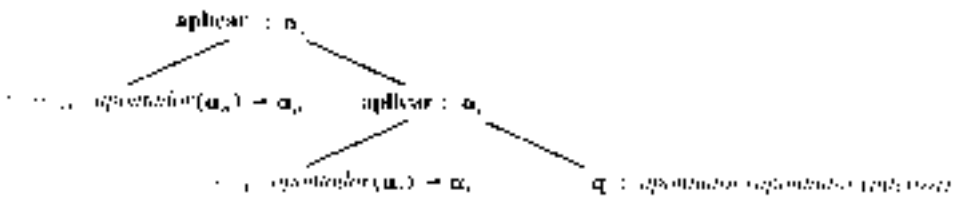


Fig. 6.17. Árvores [illegible]

6.2. Antes de apresentá-las, ilustramos essas diferenças considerando a expressão [illegible] no programa 6.6. Uma árvore sintática para esta expressão é mostrada na Fig. 6.17. Anotados a cada nó existem dois tipos. O primeiro nos informa a sub-expressão representada pelo nó e o segundo é uma expressão de tipos associada à sub-expressão. Os dois tipos são distintos em suas ocorrências mais externas e iguais [illegible] respectivamente.

As diferenças entre as regras das funções ordinárias são:

1. As ocorrências de variáveis de tipo para uma função polimórfica na mesma expressão não precisam ter argumentos do mesmo tipo. Na expressão [illegible] tem-se uma ocorrência de [illegible] indexada pela aplicação, de tal forma que [illegible] é aplicada em argumento de tipo diferente. A implementação dessa propriedade está baseada na interpretação de $\forall \alpha$ como "para qualquer tipo α". Cada ocorrência de [illegible] possui sua própria perspectiva sobre o que a variável ligada α deve significar. Por conseguinte, um nome para cada ocorrência de [illegible] é uma expressão de tipo formada pela substituição de cada α ligado por uma variável nova e não-ligada que identifica $\forall \alpha$ naquele processo. Na Fig. 6.17, as variáveis novas β e γ são usadas nas expressões de tipo atribuídas às ocorrências mais externas e mais internas de [illegible], respectivamente.

2. Uma vez que as variáveis podem aparecer em expressões de tipo, temos que ampliar a noção de equivalência de tipos. Suponhamos que τ_1 de tipo $\alpha \to \alpha$ seja aplicada a τ_2 do tipo β no lugar de simplesmente determinar a equivalência de α e β, precisamos "unificá-los". A unificação é definida abaixo e, informalmente, determina se α e β podem ser tornados estruturalmente equivalentes através da substituição de variáveis de tipo por expressões de tipo. Por exemplo, no programa 6.6 o termo rotulado **aplicar** na Fig. 6.17 é qualificado

$$operador\ \gamma \to \gamma = operador\ operando\ inteiro \to \alpha$$

e verdadeiro se γ for substituída por $operando\ inteiro \to \alpha$.

3. Necessitamos de um mecanismo para registrar o efeito de unificar duas expressões. Em geral, uma variável de tipos pode aparecer em várias expressões de tipo. Se a unificação de α e de β resultar numa variável representando o tipo γ, então é preciso continuar a representar a unidade, de tal forma que a unificação de tipos prossiga. Por exemplo, na Fig. 6.17, α é do tipo $inteiro \to \alpha$ e, dessa forma, podemos inferir para α o tipo de [illegible]. A unificação dos tipos do domínio de [illegible] com o tipo de [illegible], por conseguinte, exige a unificação de α com $\beta \to \gamma$. Isso é feito no nodo rotulado **aplicar**. A tabela variável de tipos na Fig. 6.17 representa α como $\beta \to \gamma$.

Substituições, Instâncias e Unificação

A informação sobre os tipos representados nas variáveis é formalizada definindo-se um mapeamento a partir das variáveis de tipo para as expressões de tipo, chamado de uma *substituição*. A seguinte função recursiva *substituir* toma uma expressão de tipos τ e uma substituição S para retornar a expressão de tipos que resulta da substituição em τ de cada variável de tipo α pela expressão de tipos $S(\alpha)$.

Função *substituir* (*expressão de tipos*, *substituição*)
início
  se τ é um tipo básico **então retornar** τ
  senão se τ é uma variável **então retornar** $S(\tau)$
  senão se τ é $\tau_1 \to \tau_2$ **então retornar** *substituir*(τ_1, S) $\to$ *substituir*(τ_2, S)
fim

Por conveniência, escrevemos $S(\tau)$ para a expressão de tipos que resulta quando *substituir* é aplicada a τ e a substituição S. Cada nodo de uma árvore de [illegible] associa uma expressão de tipos com uma substituição que especifica uma expressão para cada variável de tipos. Vê-se imediatamente que a substituição identidade para todas as variáveis.

Exemplo 6.10 No que se segue, escrevemos $\tau_1 \leq \tau_2$ para indicar que τ_2 é uma instância de τ_1:

$operador\ inteiro \to inteiro$	$\leq$	$operador\ \alpha \to \alpha$
$operador\ \beta \to \beta$	$\leq$	$operador\ \alpha \to \alpha$
$ponteiro \to inteiro$	$\leq$	$\alpha \to \alpha$
$operador\ inteiro \to \beta$	$\leq$	β
$\alpha \to \beta$	$\leq$	β

No entanto, ninguém se segue, o tipo de uma expressão à esquerda não é uma instância daquela à direita pela razão indicada:

$inteiro$	$\leq$	$real$	As substituições para se aplicam aos tipos básicos
$inteiro \to inteiro$	$\leq$	$\alpha \to \alpha$	Substituições inconsistentes para α
$inteiro \to \alpha$	$\leq$	$\alpha \to \alpha$	Todas as ocorrências de α devem ser substituídas

Dizemos que τ_1 e τ_2 são *unificáveis* se existir alguma substituição S tal que $S(\tau_1) = S(\tau_2)$. Na prática, estamos interessados no mais geral dos unificadores, que é uma substituição que impõe o mínimo de restrições sobre as variáveis das expressões. Mais precisamente, o unificador mais geral das expressões τ_1 e τ_2 é uma substituição S com as seguintes propriedades:

1. $S(\tau_1) = S(\tau_2)$
2. para qualquer outra substituição S' tal que $S'(\tau_1) = S'(\tau_2)$, a substituição é uma instância de S, isto é, para qualquer S' existe uma substância S'' tal que $S'(\alpha) = S''(S(\alpha))$ para todo α.

No que segue, quando dizemos "unifica", estaremos nos referindo ao unificador mais geral.

Verificando as Funções Polimórficas

As regras para a verificação de expressões gerais pela gramática da Fig. 6.16 serão escritas em termos das seguintes operações numa representação de tipos sob a forma de grafos.

1. ... substituir as variáveis ligadas na expressão de tipo para substituí-las pelo nó apontador para um nó representando a expressão de tipo resultante. Quando encontrarmos símbolos $\forall$ durante esse processo ...

2. ... unificar as duas expressões de tipos representadas pelos nós apontados por n_1 e n_2. Passar o efeito desta unificação para a substituição que torna as expressões equivalentes. Se as expressões não puderem ser unificadas, todo o processo de verificação de tipos falha.

As folhas rotuladas com variáveis de tipos são criadas utilizando-se operações como *nova_folha* e *criar_nós*, similares àquelas da Seção 5.2. É necessário que haja uma única folha para cada variável de tipo, pois outras expressões estruturalmente equivalentes são representadas pelos mesmos nós.

A operação *unificar* é realizada na seguinte forma, utilizando a técnica de unificação e substituições. Suponhamos que os nós n_1 e n_2 de um grafo representem as expressões E_1 e E_2, respectivamente. Dizer que os nós n_1 e n_2 são equivalentes sob a substituição S se $S(n_1) = S(n_2)$ é o problema de se encontrar o unificador mais geral S pode ser determinado como sendo o problema de se agrupar em conjuntos os nós que precisam ser equivalentes sob S. Para as expressões serem equivalentes suas raízes precisam ser equivalentes. Igualmente, dois nós n e n' são equivalentes se e somente se representarem o mesmo operador e seus filhos correspondentes forem equivalentes.

Um algoritmo para unificar um par de expressões será estudado na próxima seção. O algoritmo constrói os conjuntos de nós que são equivalentes sob as substituições que encontramos.

As regras de verificação de tipos para as expressões são apresentadas na Fig. 6.18. Não mostramos como as declarações são processadas. Sabemos que as expressões de tipo geradas pelos não-terminais T e Q são armazenadas em folhas e *criar_folha* é chamada para criar o nó folha do tipo, segundo a construção da GDS da Seção 5.7. Quando um identificador é declarado, o ponteiro da folha é guardado na tabela de símbolos sob a forma de um apontador para o nó que representa o tipo. Na Fig. 6.18, esse apontador é referenciado como *entrada_tabela(id.lexema)*. Como mencionado acima, *unificar* é chamada para unificar os símbolos $\forall$ à medida que substituímos as variáveis ligadas por variáveis novas. A ação associada à produção $E \rightarrow E_1\ E_2$ estabelece E como o conjunto dos tipos de E_1 e E_2.

A regra de verificação de tipos para a aplicação de funções $E \rightarrow E_1(E_2)$ é motivada pela consideração do caso onde $E_1.tipo \neq E_2.tipo$ e duas variáveis quaisquer de tipo digamos que $E_1.tipo = \alpha$ e $E_2.tipo = \beta$. Agora E_1 não pode ser uma função a não ser que para algum tipo desconhecido γ tenhamos $\alpha = (\beta \rightarrow \gamma)$. Na Fig. 6.18, uma variável nova correspondente a γ é criada e $E_1.tipo$ é unificada com $E_2.tipo \rightarrow \gamma$. Uma nova variável de tipo é criada a cada chamada de *nova_folha*, uma folha para a mesma e outra folha para γ, que não representam tipo algum. Após a unificação ter sido executada, a folha representa o tipo resultante.

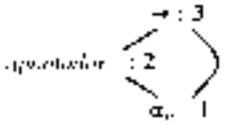
As regras da Fig. 6.18 serão ilustradas trabalhando-se sobre um exemplo simples em detalhe. Sumarizamos o funcionamento do algoritmo escrevendo a expressão de tipo construída a cada subexpressão como na Fig. 6.19. A cada aplicação de função, a operação *unificar* pode ter o efeito colateral de registrar uma expressão de tipo para algumas das variáveis de tipo. Tais efeitos colaterais são ignorados pela ênfase para a substituição na Fig. 6.19.

Exemplo 6.11. A verificação de tipos da expressão ... ... procede de baixo para cima a partir das folhas. Mais uma vez, os subscritos nos α distinguem as ocorrências de

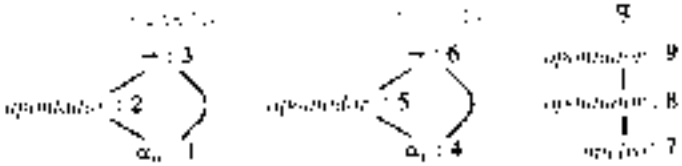
T	$\rightarrow$ $T_1 \rightarrow T_2$	{ $t = criar\_folha(criar\_operador(\rightarrow, criar\_folha(), criar\_folha()))$; $unificar(t.tipo, criar\_folha())$; $t.tipo = t$ }
T	$\rightarrow$ $T_1\ T_2$	{ $t.tipo = criar\_folha(criar\_operador(\cdot, T_1.tipo, T_2.tipo))$ }
T	$\rightarrow$ **id**	{ $t.tipo = entrada\_tabela(\textbf{id}.lexema).tipo$ }

Fig. 6.18. Esquema de tradução para a verificação de tipos polimórficos

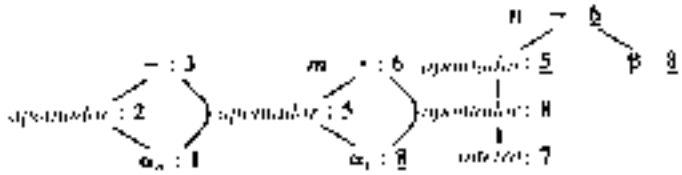
... Quando a subexpressão ... ... é chamada, novos nós são criados para uma nova variável de tipo.



O número no nó indica a classe de equivalência que o contém pelo momento. A partir do grafo de tipos para os três identificadores é mostrada abaixo. As linhas pontilhadas indicam que os nós numerados 3, 6 e 9 serão unificados ... e ..., respectivamente.



A aplicação de função ... ... é verificada pela construção de um nó para uma função a partir do tipo de ..., uma variável de tipo β e uma função que unifica seu sucessor com o tipo de ... representado pelos nós abaixo. Antes dos nós serem unificados, cada um possui um número distinto. Após a unificação, os nós equivalentes são aqueles abaixo com o mesmo conjunto de números anotados sobrepostos.



Note que o operador *operador interno*, são ambos numerados 5, pois eles são unificados com sua subexpressão de tipo correspondente na Fig. 6.19. Subsequentemente, ... é unificada com *inteiro*. ☐

O próximo exemplo relaciona a inferência de tipos de funções polimórficas em ML para as regras de verificação de tipos na Fig. 6.18. A sintaxe das definições de funções em ML é dada por

$$\textbf{fun}\ \textbf{id}_0\ \textbf{id}_1\ \textbf{id}_2\ \ldots\ \textbf{id}_n = E\ .$$

onde $\textbf{id}_0$ representa o nome da função e $\textbf{id}_1, \ldots, \textbf{id}_n$ representam seus parâmetros. Para simplificar, assumimos que a sintaxe da expressão E é como na Fig. 6.1 (por que os únicos identificadores em E são os nomes da função, de seus parâmetros e de funções embutidas na linguagem).

A abordagem é uma formalização do Exemplo 6.9, onde o tipo polimórfico do índice para ... ... Novas variáveis de tipo são

EXPRESSÃO	TIPO	SUBSTITUIÇÃO
	operação sobre variável de controle	
	operação sobre	
	operação sobre	$\alpha = \textit{operando inteiro}$
	operação sobre	
	valor	$\alpha_1 = \textit{booleano}$

Fig. 6.19. Sumário de computação de tipos durante a verificação

estabelecidas para o nome da função e seus parâmetros. De um modo geral, as funções embutidas têm tipos polimórficos, com um ou mais tipos de que dependem aqueles tipos estão ligados a parâmetros de tipo. Verificações semelhantes são feitas para os tipos das expressões `id₁ id₂` e `id₁ id₂ id₃`, e as correspondentes. Quando uma correspondência ocorre, temos uma instância de unificação para um nome de função. Igualmente, quaisquer variáveis de tipo inteiro são amarradas por quantificadores ∀ para fornecer o tipo polimórfico para a função.

Exemplo 6.12. Relembre a função `MAP` na Fig. 6.14 para determinar o comprimento de uma lista

```
                    .
                    .
                    .
```

As variáveis de tipo `β` e `γ` são instanciadas para os tipos de `` e `` respectivamente. Uma instância apropriada de `` corresponde ao tipo da expressão que forma o corpo da função e que `` determinou o tipo

```
para qualquer tipo booleano
```

e dessa forma o tipo de `` é

```
Var booleano → inteiro
```

Mais detalhadamente, a verificação do programa mostrado na Fig. 6.20, para os tipos das regras de verificação de tipos da Fig. 6.18, podem ser aplicadas. As declarações no programa associam novas variáveis de tipo `β` e `γ` a `` e `` e informam explicitamente os tipos das funções embutidas. Escrevemos condicionais no estilo da Fig. 6.16 pela aplicação da operação polimórfica `` a três operandos representando a expressão a ser testada, a parte `then` e a parte `else`. A declaração diz que as partes `then` e `else` podem ser de quaisquer tipos que se unifiquem, que será também o tipo do resultado.

Claramente `` precisa ser verificada antes que o corpo da função, essa verificação é realizada usando a especificação

```
                    :
                    :
                    : ∀α, β. Var α → Var β → Var α
                    : ∀α. Var booleano → α → α → α
                    : ∀α. Var booleano → booleano
                    : inteiro
                    : inteiro
                    : ∀α, β. Var α → Var β → booleano
                    : ∀α. Var α → booleano
                    .
                    .
                    .
```

Fig. 6.20. Declarações usadas pela expressão a ser verificada

acima. Uma vez que a função `` é chamada com uma lista de inteiros, que permite que toda a verificação seja feita usando apenas o fragmento do exemplo da Fig. 6.18.

O efeito da aplicação das regras de verificação de tipos da Fig. 6.18 ao programa na Fig. 6.20 é sumarizado na Fig. 6.21. As novas variáveis introduzidas pela operação `` e aplicadas aos tipos polimórficos das operações embutidas são distinguidas por subscritos em α. Aparecem na linha 3 que `` precisa ser uma função de γ para algum γ pode-se concluir. Em segunda, quando as expressões são `` foi verificada encontramos na linha 6 que `` precisa ser uma função `` onde `` é um tipo desconhecido. A esse ponto sabemos que o tipo de `` precisa ser

```
para qualquer tipo booleano
```

Escrevemos aqui, que a aplicação verificada a tipo `` é para a instância de escrever `` entre os argumentos por uma questão de clareza `` é um método com *notação*.

Quando a verificação estiver concluída, a variável de tipo permanece no tipo de `` . Como nenhuma suposição foi feita sobre ela, a mesma pode ser substituída por qualquer tipo que a função for usada. Consequentemente, tratamos-a como uma variável ligada e escrevemos

```
∀α. booleano → α → α → α
```

para o tipo de `` .

6.7 UM ALGORITMO PARA A UNIFICAÇÃO

Informalmente, unificação é o problema de determinar se duas expressões e_1 e e_2 podem ser tornadas idênticas pela substituição das variáveis por expressões em e_1 e e_2. Teste de igualdade de expressões é um caso especial da unificação, se e_1 e e_2 possuem constantes, mas não variáveis então e_1 e e_2 são unificáveis somente se forem idênticas. O algoritmo de unificação nesta seção pode ser aplicado a grafos com ciclos, e por conseguinte pode ser usado para testar expressões recursivas dos tipos unitários.[1]

A unificação foi definida na última seção em termos de uma função S, chamada de uma substituição, mapeando variáveis em expressões. Escrevemos Se para a expressão obtida quando cada variável em e é substituída por Sv. Se u é um unificador para e_1 e e_2, $Se_1 = Se_2$. O algoritmo desta seção determina a substituição que é o unificador mais geral para um par de expressões.

Exemplo 6.13. Para uma pesquisa sobre os unificadores mais gerais, consideremos as duas expressões de tipos

```
Var α → Var α → booleano
Var β → Var γ → booleano
```

[1] Um algoritmo eficiente para a unificação de expressões recursivas é dado por Huet, G. P. "Resolution d'equations dans les langages d'ordre 1, 2, ..., ω", Thèse d'Etat, Université de Paris VII, 1976.

Índice	Expressão	Tipos	Substituição
(1)	[illegible]	γ	
(2)	[illegible]	β	
(3)	[illegible]	δ	$\beta = \gamma \to \delta$
(4)	[illegible]	γ	
(5)	[illegible]	*lista de* $\alpha_1 \to$ *lista de inteiros*	
(6)	[illegible]	*lista de inteiros*	$\gamma =$ *lista de* α_1
(7)	[illegible]	*inteiro*	
(8)	[illegible]	*lista de* α_1	
(9)	[illegible]	*lista de* $\alpha_1 \to$ *lista de inteiros*	
(10)	[illegible]	*lista de* α_1	$\alpha_1 = \alpha_1$
(11)	[illegible]	*lista de* $\alpha_1 \to \alpha$	
(12)	[illegible]	δ	
(13)	[illegible]	*inteiro*	
(14)	[illegible]	*inteiro* $\to$ *lista de inteiros* $\to$ *lista de inteiros*	
(15)	[illegible]	*inteiro*	$\alpha = inteiro$
(16)	[illegible]	*lista de inteiros* $\to \alpha \to \alpha_1 \to \alpha_1$	
(17)	[illegible]	*inteiro*	$\alpha_1 =$ *inteiro*
(18)	[illegible]	$\alpha_{18} \to \alpha_{18} \to \alpha_{18}$	
(19)	[illegible]	*inteiro*	$\alpha_{18} =$ *inteiro*

Fig. 6.21. Interseção [illegible] para [illegible].

Dois unificadores S_1 e S_2 para essas expressões são:

x	$S_1(x)$	$S_2(x)$
α	α_1	α_1
β	α_1	α_1
γ	α_1	α_1
α_1	α_1	α_1
α	*lista de inteiro*	*lista de* α_1

pois substituições importantes são como segue:

$$S_1(\alpha) = \alpha_1(\beta = \alpha_1 \to \alpha_1) = \text{lista de inteiros} \to \text{lista de inteiros}$$
$$S_2(\alpha) = \alpha_1(\beta = \alpha_1 \to \alpha_1) = \text{lista de } \alpha_1 \to \text{lista de inteiros}$$

A substituição S_2 é unificador mais geral para e e f. Note-se que S_1 é uma instância de S_2 porque S_1 substitui α_1 por *inteiro* em ambas as variáveis em S_2 que contêm α_1. O inverso, no entanto, é falso, pois cada ocorrência de α_1 em S_2 pode ser substituída pela mesma expressão dessa forma não podemos obter S_2 a partir de S_1 substituindo a variável α em $S_1(\alpha)$.

Quando as expressões a serem unificadas representam árvores, o número de nós na árvore para a expressão substituída $S(e)$ pode ser o exponencial do número de nós nas árvores para e e f, mesmo que S seja o unificador mais geral. Infelizmente, tal explosão de tamanho pode ocorrer se formos usados grafos em lugar de árvores para representar expressões e substituições.

Temos implementado uma formulação grafo-teorética da unificação, também apresentada na última seção. O problema é o de agrupar em conjuntos os nós que precisam ser equivalentes sob a unificação para geral de duas expressões. As duas expressões do Exemplo 6.13 são representadas pelos dois nós rotulados $\to$ na Fig. 6.22. Os outros nós nos dão indicações das classes de equivalência às quais os nós pertencem após os nós raízes terem sido unificados. Essas classes de equivalência possuem a propriedade de todos os nós interiores numa classe de equivalência serem para expressões operadas. Os filhos correspondentes destes nós interiores numa classe de equivalência são também equivalentes.

Algoritmo 6.1. Unificação de um par de nós numa grafo.

Entrada. Um grafo e um par de nós m e n a serem unificados.

Saída. O valor *booleano* verdadeiro se as duas expressões representadas pelos nós m e n se unificarem e falso em caso contrário. A versão da operação *encontre* necessária para as regras de verificação de tipos da Fig. 6.16 é rotulada se a função neste algoritmo é modificada para falhar ao invés de retornar falso.

Método. Um nó é representado por um registro, como na Fig. 6.23, com campos para o operador binário e apontadores para os filhos à esquerda e à direita.

Os conjuntos de nós equivalentes são mantidos usando-se o campo *próximo*. Um nó em cada classe de equivalência é escolhido como o único elemento representativo da classe de equivalência fazendo-se seu campo *conjunto* conter o apontador nil. Os campos *conjunto* dos

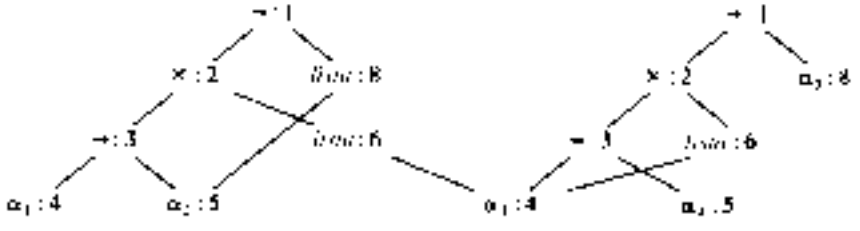


Fig. 6.22. Classes de equivalência após unificação.

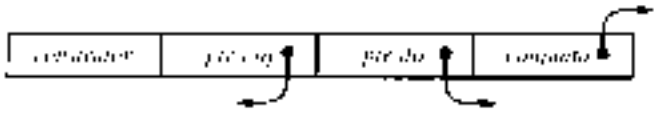


Fig. 6.23 Um conjunto de dados para unificação

nos retornam os elementos representativos das classes de equivalência correspondentes aos tipos dados. Indiretamente através desses pontos nos encontramos para o elemento representativo da classe de equivalência para a classe de equivalência a que pertence o próprio α como elemento representativo.

O algoritmo de unificação, mostrado na Fig. 6.24, usa as seguintes operações sobre os nós:

1. *encontre*(α) retorna o nó representativo da classe de equivalência contendo atualmente o nó α.

2. *combine*(α, β) combina as classes de equivalência contendo os nós α e β. Se os dois elementos representativos para as classes de equivalência de α e β forem os mesmos nó, então nenhuma operação será necessária, pois α e β já pertencem à mesma classe de equivalência. Caso contrário, *combine* escolhe um dos dois elementos representativos como o novo representante. Essa assimetria na especificação de *combine* é importante porque uma variável não pode ser usada como o elemento representativo de uma classe de equivalência para uma expressão contendo um construtor de tipos ou um tipo básico. De outra forma, duas expressões não equivalentes poderiam ser unificadas através da variável.

A operação de *união* sobre conjuntos é implementada simplesmente mudando-se um dos ponteiros do elemento representativo de cada uma das classes de equivalência de forma que o mesmo aponte para o elemento representativo da outra. Para encontrar a classe de equivalência a que um nó pertence, seguimos os apontadores começando desse nó até que o elemento representativo (o nó que contém a si mesmo) seja alcançado.

Note que o algoritmo da Fig. 6.24 usa $\alpha = encontre(\alpha)$ e $\beta = encontre(\beta)$ em vez de α e β respectivamente. Se os dois representativos α e β representarem o mesmo tipo básico, a chamada *mesmo?*(α, β) retorna verdadeiro. Se α e β são ambos nós internos para um construtor binário de tipos, combinamos as suas classes de equivalência (uma base especulativa) e verificamos recursivamente se seus filhos respectivos são equivalentes. Combinando as primeiras duas variáveis de classe de equivalência antes de verificar recursivamente os filhos e dessa forma o algoritmo termina.

A substituição de uma variável por uma expressão é implementada pela adição da folha para a variável à classe de equivalência que contém a folha para a expressão. Se uma das duas classes contém uma folha, isso é folha para uma variável que tenha sido colocada numa classe de equivalência que contém também um nó que representa uma expressão com um construtor de tipos ou com um tipo básico, *substitua* irá retornar um elemento representativo que será o construtor de tipos ou tipo básico.

```
função mesmo?(α, β : nós) : booleano
início
    α := encontre(α)
    β := encontre(β)
    se α = β então
        retornar verdadeiro
    senão se α e β são nós que representam o mesmo tipo básico
        então retornar verdadeiro
    senão se α e β são nós de operadores com filhos (α₁, α₂) e (β₁, β₂)
            e o nó de operador com filhos (β₁, β₂) é então início
        combine(α, β);
        retornar mesmo?(α₁, β₁) e mesmo?(α₂, β₂)
    fim
    senão se α e β representam tipos básicos então início
        combine(α, β);
        retornar verdadeiro
    fim
    senão retornar falso
        {os nós internos com operadores diferentes não podem
        ser unificados}
fim
```

Fig. 6.24. Algoritmo de unificação

Dizemos que uma variável não pode ser unificada a duas expressões diferentes. □

Exemplo 6.14. Mostramos o passo inicial para duas expressões de tipo $(1 \times 2) \to 3$ e $(4 \to 5) \to 6$ em conjunto com os nós em uma mesma classe de equivalência. Para computar *unificar*$(1, 4)$, o algoritmo nota que os nós 1 e 4 representam o mesmo operador, portanto combina 1 e 4 em uma mesma classe de equivalência e chama *unificar*$(2, 5)$ e *unificar*$(3, 6)$. O resultado de computar *unificar*$(1, 4)$ é ilustrado na Fig. 6.22. □

Se o Algoritmo 6.1 retornar verdadeiro, poderemos construir uma substituição S que associa com um unificado. Examinamos cada nó original para ver se ele é equivalente a uma expressão associada com S. Dessa forma, para cada variável β, *encontre*(β) nos fornece o elemento representativo da classe de equivalência de β. A expressão representada por $S\beta$ é β se *encontre*$(\beta) = \beta$. Por exemplo, na Fig. 6.22 vemos que 9 é o elemento representativo para 9 e 10 e 14, que representam α_3. O elemento representativo para 11 é o nó 8, que representa $\beta \to \beta$.

Exemplo 6.15. O Algoritmo 6.1 pode ser usado para testar a equivalência estrutural das duas expressões de tipo

$$(\alpha \times \beta) \to \gamma$$
$$(\delta \to \delta) \to (\alpha \to \gamma)$$

Os grafos de tipo para essas expressões são mostrados na Fig. 6.26. Uma ordenação conveniente também foi numerada.

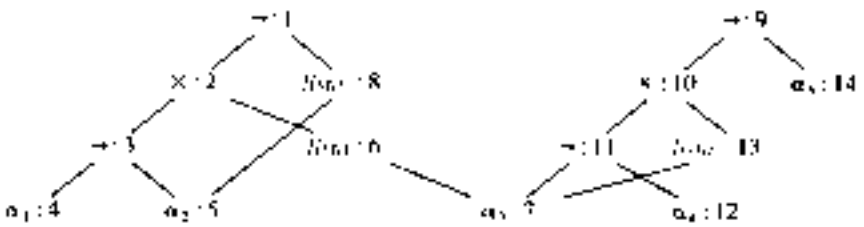


Fig. 6.25. Os nós 1 e 4 unificados com seu pai e a classe de equivalência

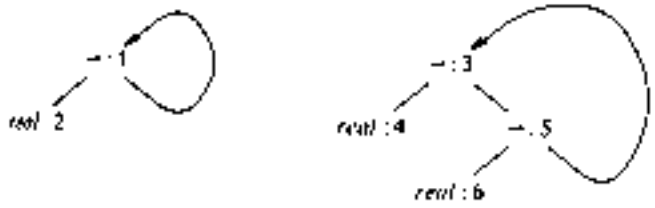


Fig. 6.26. Grafos para dois tipos recursivos.

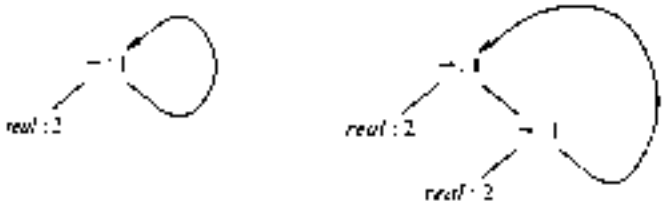


Fig. 6.27. Grafos mostrando duas classes de equivalência infinitas.

Chamamos *mesmo*(1, 3) para testar a equivalência estrutural dessas duas expressões. O algoritmo combina os nós 1 e 3 em uma única classe de equivalência e recursivamente chama *mesmo*(2, 4) e *mesmo*(5, 5). Uma vez que 2 e 4 representam o mesmo tipo básico, a chamada *mesmo*(2, 4) retorna verdadeiro. A chamada *mesmo*(5, 5) coloca 5 à classe de equivalência de 1 e de 3 e chama recursivamente *mesmo*(2, 6) e *mesmo*(5, 1).

A chamada *mesmo*(2, 6) retorna verdadeiro porque 2 e 6 também representam o mesmo tipo básico. A segunda chamada de *mesmo*(5, 1) termina porque já havíamos combinado os nós 1 e 3 na mesma classe de equivalência. O algoritmo então retorna verdadeiro para mostrar que as duas expressões são de fato equivalentes. A Fig. 6.27 mostra as classes de equivalência de nós resultantes, onde os nós com o mesmo número inicial estão na mesma classe de equivalência.

EXERCÍCIOS

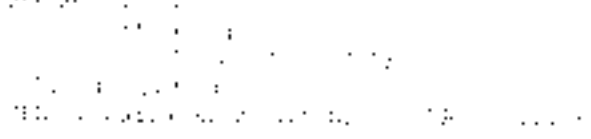
6.1 Escreva expressões de tipo para os seguintes tipos:

a) Um *array* de apontadores de reais, onde os índices do *array* variam de 1 a 100.

b) Um *array* bidimensional de inteiros tal que, um *array* de *arrays* cujos índices sejam independentes de 0 a 9 e as colunas de −10 a 10.

c) Funções cujos domínios sejam funções de inteiros para apontadores de inteiros e cujos intervalos sejam registros envolvendo um inteiro e um caractere.

6.2 Vamos supor que temos as seguintes declarações C:



Escreva expressões para os tipos [illegible].

6.3 A seguinte gramática define listas de listas de literais. A interpretação dos símbolos é a mesma que a da gramática da Fig. 6.3 com a adição do tipo **list**, que indica uma lista de elementos do tipo *T* que o segue.

$$
\begin{aligned}
T &\rightarrow \mathbf{list}\ \mathbf{of}\ T \mid \mathbf{char} \mid \mathbf{integer} \\
L &\rightarrow \mathit{literal} \mid \mathbf{num} \mid \mathbf{id}
\end{aligned}
$$

Escreva regras de tradução semelhantes àquelas da Seção 6.2 para determinar os tipos das expressões formadas pelas listas de literais.

6.4 Adicione a gramática do Exercício 6.3 a produção

$$E \rightarrow \mathbf{nil}$$

significando que uma expressão pode ser a lista vazia. Revise as regras de sua resposta ao Exercício 6.2 de forma a levar em conta o fato de que **nil** pode figurar em lugar de uma lista vazia de elementos de qualquer tipo.

6.5 Usando o esquema de tradução da Seção 6.2, compute os tipos das expressões nos seguintes fragmentos de programa. Mostre os tipos em cada nó da árvore gramatical.

a) [illegible]

b) [illegible]

c) [illegible]

6.6 Modifique o esquema de tradução para verificação de expressões na Seção 6.2 de forma a que suprima uma mensagem de erro após ter detectado o primeiro erro, a menos que o tipo esperado tivesse sido convertido.

6.7 Reescreva as regras de verificação de tipos para expressões na Fig. 6.2 de forma a que se refiram aos nós de uma representação sintática para tipos em vez de tipos. As regras resultantes devem usar estruturas de dados e operações suportadas por uma linguagem como Pascal. Use a equivalência natural para as expressões de tipo quando:

a) as expressões de tipo forem representadas por árvores, como na Fig. 6.2; e

b) o grafo de tipos for um GDA com um nó/nós para cada expressão de tipo.

6.8 Modifique o esquema de tradução da Fig. 6.5 para tratar o que se segue:

a) Comandos que possuam valor. O valor de um comando de atribuição é o da expressão à direita do símbolo =. O valor de um comando condicional ou de repetição **while** é o valor do corpo do comando, e o valor de uma lista de comandos é o valor do último elemento da lista.

b) Expressões booleanas. Adicione produções para os operadores lógicos **and**, **or** e **not** e para os operadores de comparação <, <=, =, <> (não igual), >= e >. Em seguida, adicione as regras de tradução apropriadas para as expressões de tipo.

6.9 Generalize as regras de verificação de tipos para as funções dadas ao final da Seção 6.2 para tratar funções recursivas.

6.10 Assuma que os nomes de tipos [illegible] sejam definidos como na Seção 6.3. Quais das seguintes expressões são estruturalmente equivalentes?

i) [illegible]

ii) *apontador* [illegible]

iii) *apontador* [illegible]

iv) *apontador* [illegible] *array* [illegible]
apontador [illegible]

6.11 Reorganize o algoritmo para testar equivalência estrutural na Fig. 6.6, de forma que os argumentos de *unify* sejam apontadores para nós de um GDA.

6.12 Considere a codificação de expressões restringidas de tipos como seqüências de *bits* no Exemplo 6.1. Em Johnson [1979], os campos de dois bits para os construtores apareceram na ordem oposta, com o campo para o construtor mais externo ocupando os bits menos significativos. Qual é o tipo básico que, por exemplo,

Expressão de Tipo	Codificação
array of array	0000 0000 0000
pointer to pointer to	0000 1100 0000
operation from array to array	1001 1111 0000
operation from pointer to pointer to	1101 1111 0000

Usando os operadores de C, escreva um código para construir a representação do tipo em a partir daquela de duas seqüências, assumindo que a codificação esteja em

a) Johnson [1979]

b) Exemplo 6.1.

6.13 Suponhamos que o tipo do identificador seja um subintervalo dos inteiros. Para expressões com os operadores +, −, ×, ÷, ↑ e −, como em Pascal, escreva regras de verificação de tipos que avisem, caso uma subexpressão ou o intervalo em que o valor pertença estar contido.

6.14 Dê um algoritmo para testar a equivalência dos tipos de C vistos no exemplo 6.11.

6.15 Algumas linguagens, como PL/I, não exigem um valor *boolean* mas um valor inteiro com convenções identificado por 1 e falso por 0. Por exemplo, *3<5* é agrupado como *bool(3<5)*, e tem o valor "verdadeiro" com 1, porque *3<5* tem o valor 1 e *1=5* é verdadeiro. Escreva regras de tradução para expressões *boolean* que leve em conta essa convenção. Utilize instruções condicionais da linguagem intermediária para converter valores inteiros a temporários que representem o valor de uma expressão *boolean* quando necessário.

6.16 Generalize os algoritmos das Fig. 6.9 e 6.10 e 6.12 para expressões com construtores de tipos arranjo, apontador e produto cartesiano.

6.17 Quais das seguintes expressões de tipo recursivo não são equivalentes?

pointer to −
array from − *integer to* −
pointer to → *integer* → −

6.18 Usando as regras do Exemplo 6, determine quais das seguintes expressões possuem tipos únicos. Assuma que *c* seja um número complexo.

a) −
b) −
c) −

6.19 Suponhamos que sejam permitidas conversões de tipos no Exemplo 6.6. Sob que condições sobre os tipos de *x*, *y* e *z* o tipo resultante contém todas as expressões *x* × (*y* + *z*) com um único tipo?

6.20 Expresse, usando variáveis de tipo, os tipos das seguintes funções:

a) A função *swap* que troca os valores apontados por dois objetos de qualquer tipo e retorna um apontador para o objeto.

b) Uma função que toma como argumentos um *array* indexado pelos inteiros, com elementos de qualquer tipo, e retorna um par cujos elementos sejam os objetos apontados pelos elementos do *array* dado.

6.21 Encontre o unificador mais geral para as seguintes expressões de tipo:

i) *expression from* × *to* →
ii) β × (γ → δ)
Como seria se um dos tipos fosse *int*?

6.22 Encontre o unificador mais geral para cada par de expressões da lista seguinte ou estabeleça que nenhum existe.

a) $\alpha_1 \to (\alpha_2 \to \alpha_3)$
b) *array* (α_1, α_2) *of* *expression from* $(\alpha_3) \to \beta_1$
c) $\gamma \to \gamma$
d) $\delta_1 \times \delta_2 \to \delta_3$

6.23 Expanda as regras de verificação de tipos no Exemplo 6.10 para atender registros. Use a seguinte sintaxe geral para expressões de tipo e expressões.

$$\begin{aligned} T &\to \textbf{record} \; \textit{campos} \; \textbf{end} \\ E &\to E \; \textbf{id} \\ \textit{campos} &\to \textit{campos} \; ; \; \textit{campo} \mid \textit{campo} \\ \textit{campo} &\to \textbf{id} \; : \; T \end{aligned}$$

Que restrições a ausência de nomes de tipos impõe sobre os tipos que podem ser definidos?

***6.24** A resolução para a sobrecarga na Seção 6.5 procede através de duas fases: primeiro o conjunto de tipos possíveis para cada subexpressão é determinado e, em segunda, estendendo nesta segunda fase para um único tipo. Apesar de tipos ainda poderem ser não determinados. Que estruturas de dados você usaria para resolver a sobrecarga numa única passagem de baixo para cima (*bottom-up*)?

****6.25** A resolução da sobrecarga se torna mais difícil se forem declarações de identificadores como operadores. Mas precisamente, suponhamos que as declarações possam ser usadas para sobrecarregar identificadores que representem símbolos de funções, mas que todas as ocorrências de um identificador num bloco tenham o mesmo tipo. Mostre que o problema de determinar se uma expressão nesta linguagem possui um tipo válido é NP-completo. Este problema emerge durante a verificação de tipos na linguagem experimental Hope (Burstall, MacQueen, Sannella [1980]).

6.26 Segundo o Exemplo 6.12, infira o seguinte tipo polimórfico para *twice*:

twice : ∀α ∀β (α → β) × (*array* of α) → *list* of β

A definição ML de *twice* é

```
let twice(f, a) =
    let g i = f(a[i])
    in [g 0, g 1, g 2, g 3, g 4]
```

Os tipos das identificações embutidas no corpo da função são

+	:	∀α	*integer* × *integer* → *integer*
[, , , ,]	:	∀α	α × α × α × α × α → *list* of α
a[ ]	:	∀α	(*integer* × *array* of α) → α
f( )	:	∀α	*integer* → α
g	:	∀α	*integer* → *integer*

****6.27** Mostre que o algoritmo de unificação da Seção 6.7 determina o unificador mais geral.

***6.28** Modifique o algoritmo da Seção 6.7 de tal forma que o mesmo unifique uma certa classe de expressões contendo apenas variável.

****6.29** Suponhamos que as expressões sejam representadas por árvores. Encontre as expressões *e* e *f* tais que para qualquer unificador *S* o número de nós em *Se* seja uma exponencial mínima do número de nós em *Sf*.

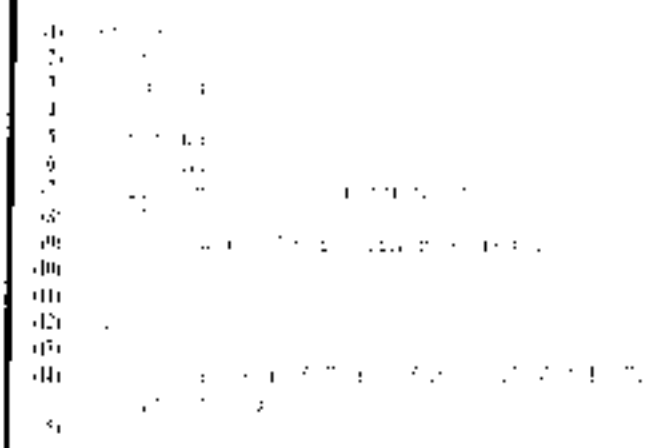


Fig. 6.28. Um programa e o resultado da tipação de uma função auto-aplicada.

6.30 Dois nós sintáticos congruentes representam expressões equivalentes. Mostre que não existem pares de nós congruentes no grafo original de tipos, após a unificação, é possível que nós distintos sejam congruentes.

a) Dê um algoritmo para combinar uma classe de nós mutuamente congruentes num único nó.

b) Escreva o algoritmo em Lisp para combinar nós congruentes até que não haja dois nós distintos que sejam congruentes.

6.31 A expressão `(= x x)` na linha 9 do programa C completo na Fig. 6.28 é a aplicação de uma função a si mesma. A declaração à linha 3 força `=` a ter o tipo `(int int -> int)`, mas os tipos dos argumentos de `x` não são especificados. Tente executar o programa. O compilador deve emitir uma advertência porque `x` é declarado como uma função à linha 3 em lugar de um apontador para uma função.

a) O que Wand pode dizer a respeito do tipo de `x`?

b) Use as regras de verificação de tipos para as funções polimórficas na Fig. 6.18 para inferir um tipo para `x` no programa a seguinte:

```
x(x)
int x(int y)
{
    return y == x(y);
}
```

NOTAS BIBLIOGRÁFICAS

Os tipos básicos e os construtores de tipos das primeiras linguagens como Fortran e Algol 60 eram limitados o suficiente para que a verificação de tipos não constituísse um problema sério. Outros resultados sobre descrições da verificação de tipos em seus compiladores são especialmente nas discussões da geração de código para as expressões. Sammet [1969] descreve a tradução de expressões no compilador Fortran original. O compilador controla se o tipo de uma expressão era inteiro ou real, mas as coerções não eram permitidas pela linguagem. Backus [1981, p. 34 tradução] observa: "Deixe que justamente porque não gostávamos das regras sobre o tipo dos resultados das expressões em modo misto decidimos: 'Vamos pagar pelo É erros [sic]'". Naur [1963] é uma exceção primordial na verificação de tipos num compilador Algol. As técnicas usadas pelo compilador são similares àquelas descritas na Seção 6.2.

Estudos de estruturação de dados tais como vetores e registros foram antecipados nos anos 40 por Zuse em seu Plankalkül, que teve pouca influência direta (Bauer e Wössner [1972]). Uma das primeiras linguagens de programação a permitir que expressões de tipos fossem construídas sistematicamente é Algol 68. Duas expressões podem ser recursivamente definidas e a equivalência estrutural é usada. Uma distinção clara entre a equivalência estrutural e a equivalência por nome é encontrada em PL/I, a escolha sendo deixada para o programador (Wegbreit [1974]). A Crítica de Parnas por Welsh, Sneeringer e Hoare [1977] chamou a atenção para a discussão.

A combinação de coerção e sobrecarga pode levar a ambiguidades exageradas argumentos pela resolução da sobrecarga a favor de um algoritmo diferente. As restrições são, por conseguinte, colocadas numa ou noutra. Uma abordagem liberadora da criação foi tomada em PL/I, onde o primeiro critério de projeto era "*Qualquer coisa pode*. Se uma combinação particular de símbolos possuir um significado razoável, esse significado será tomado". Infelizmente, frequentemente o significado razoável é o errado. Um entendimento mais completo do conjunto de tipos básicos — por exemplo, Bird [1967] descreve uma estrutura hierárquica proposta para os tipos básicos de CPL e os tipos mais sérios podem ser exigidos para os tipos mais altos.

A resolução em tempo de compilação de sobrecarga em linguagens fortemente tipadas como APL (Iverson [1962]) e SETL (Schwartz [1975]) é uma potencial de melhoria de tempo de execução dos programas (Baumann e Saal [1973]). Tennenbaum [1974] distingue entre a resolução "à frente" que determina os conjuntos de possíveis tipos de um operador, partindo de seus operandos e a resolução "para trás" baseada no tipo esperado pelo contexto. Usando um reticulado de tipos, Ichbiah e Muchnick [1976] e Kaplan e Ullman [1980] resolvem restrições sobre tipos obtidas a partir de análises à frente e para trás. A sobrecarga em Ada pode ser resolvida fazendo-se uma única passagem à frente e com segunda uma única passagem para trás como na Seção 6.3. Essa observação emerge de um número de artigos: Ganzinger e Ripken [1980], Demello, DeRemer e Meyers [1980]; Lamas [1980]; Persch et al. [1980], Gonzales [1981] descreve uma implementação recursiva e Baker [1982] exige uma passagem para trás explicitamente usando um CDA das possíveis tipos.

A inferência de tipos foi estudada por Curry e por Curry e Feys [1958] em conexão com a lógica combinatória e o cálculo lambda (*Lambda Calculus*) de Church [1941]. Tem sido frequentemente observado que o cálculo lambda é análogo às linguagens funcionais. Tennent repetidamente usado a aplicação de uma função a um argumento para discutir elementos de verificação de tipos neste capítulo. As regras podem ser derivadas e aplicadas independentemente dos tipos no cálculo lambda e Curry estava interessado em seus "caracteres funcionais" e em determinar o que chamaríamos agora de tipo polimórfico mais geral consistentes com uma expressão de tipo combinatória ou lambda. Hindley [1969] observou que a unificação poderia ser usada para inferir os tipos. Independentemente, em sua tese, Morris [1968a] atribuiu tipos a expressões lambda inventando uma série de equações e resolvendo-as de forma a determinar os tipos associados às variáveis. Desenvolvido do trabalho de Hindley, Milner [1978] também observou que a unificação pode ser usada para resolver o conjunto de equações e aplicou a idéia de inferir tipos na linguagem de programação ML.

A primeira descrição da verificação de tipos em ML é devida a Cardelli [1984]. Esta abordagem tem sido aplicada a uma linguagem de Meertens [1983]; Suzuka [1985] explora sua aplicação em Smalltalk; Wand [1986] e Ingalls [1978]; Mitchell [1984] mostra como as coerções poderiam ser incluídas.

Morris [1968a] observa que os tipos recursivos são necessários para permitir que os tipos sejam atribuídos para expressões contendo a aplicação de uma função a si mesma. O programa C da Fig. 6.28, contendo uma aplicação de uma função a si mesma, é ilustrado por um programa Algol em Ledgard [1971]. O Exercício 6.31 é proveniente de MacQueen, Plotkin e Sethi [1984], onde a validade do modelo semântico para tipos polimórfico recursivo. Enfoques diferentes aparecem em McCracken [1979] e Cardelli [1985]. Reynolds [1980] pesquisa o sistema de tipos ML, fornece diferentes teorias para evitar anomalias envolvendo coerções, sobrecarga e função polimórficas de mais alta ordem.

A unificação foi primeiramente estudada por Robinson [1965]. O algoritmo de [Martelli & Montanari 1982] pode ser prontamente adaptado a partir de algoritmos para testar a equivalência de DFs unitários. Listas ligadas com cabeça (cf. Knuth [1973c], Seção 2.2.5, Exercício 11). O algoritmo quase linear para testar a equivalência de autômatos finitos devido a [Hopcroft & Karp 1971] pode ser visto como uma implementação do esboço à pág. 584 de Knuth [1973c]. Através do uso adequado de estruturas de dados, algoritmos lineares para essa atividade são apresentados por Paterson & Wegman [1978] e por Martelli & Montanari [1982]. Um algoritmo para encontrar os mais gerais unificantes (ver Exercício 6.30) aparece em Downey, Sethi & Tarjan [1980].

Despeyroux [1984] descreve um gerador de verificadores de tipos que originou os cálculos aqui de padrões para construção verificada de tipos a partir de uma especificação semântica operacional baseada em regras de inferência.

Capítulo 7

AMBIENTES EM TEMPO DE EXECUÇÃO

Antes de uma execução de código, precisam ser feitos vários preparativos estáticos de programas e ações que precisam ocorrer em tempo de execução para implementar o programa. À medida que a execução prossegue, o mesmo nome pode denotar diferentes objetos de dados em momentos diferentes. Este capítulo examina os relacionamentos entre os nomes e os objetos de dados.

As ações de liberação de objetos de dados são gerenciadas pelo *gerenciador de armazenamento* em execução combinado com rotinas escritas pelo compilador gerado. O projeto do pacote de suporte em tempo de execução é influenciado pela semântica dos procedimentos. Pacotes de suporte para linguagens como Fortran, Pascal e Lisp podem ser razoavelmente diferentes, refletindo as diferenças nessas linguagens.

Cada execução de um procedimento é referenciada como uma *ativação* do procedimento. Se o procedimento for recursivo, várias de suas ativações podem estar vivas ao mesmo tempo. Cada chamada de procedimento em Pascal cria uma ativação que pode manipular objetos de dados alocados para ela mesma.

A representação de um objeto de dados em tempo de execução é determinada pelo seu tipo. Frequentemente os tipos de dados elementares, tais como caracteres, inteiros e reais, podem ser representados por objetos de dados equivalentes na máquina alvo. No entanto, agregados de dados tais como registros, cadeias de caracteres e estruturas são normalmente representados por coleções de objetos de dados primitivos, suas disposições são estudadas no Capítulo 8.

7.1 TEMAS DA LINGUAGEM-FONTE

Para simplificar, suponhamos que um programa seja composto por procedimentos escritos em Pascal. Esta seção distingue o texto-fonte de um procedimento de suas ativações em tempo de execução.

Procedimentos

Uma *definição de procedimento* é uma declaração que, em sua forma mais simples, associa um identificador a um enunciado. O identificador é o *nome do procedimento* e o enunciado é o *corpo do procedimento*. Por exemplo, o código Pascal na Fig. 7.1 consiste na definição do procedimento readarray, cujo corpo é dado nas linhas 3-7, e do corpo do procedimento principal nas linhas 9-25. Em algumas linguagens, os procedimentos que retornam valores são chamados de *funções*; subprogramas que não retornam valores são chamados de procedimentos. Um programa completo também será tratado como um procedimento.

Quando um nome de procedimento aparece dentro de um enunciado executável, dizemos que o procedimento é *chamado* pelo nome. A idéia básica é que uma chamada de procedimento executa o corpo do mesmo. O programa principal nas linhas 21-25 da Fig. 7.1 chama o procedimento readarray na linha 23 e em seguida chama quicksort na linha 24. Veremos que as chamadas de procedimentos podem ocorrer dentro de expressões, como na linha 16.

Alguns dos identificadores que aparecem numa definição de procedimento são especiais e são chamados de *parâmetros formais* (ou simplesmente, *formais*) do procedimento. Os chamados de *argumentos formais* e na linha 12 são os parâmetros formais de readarray. Argumentos, também chamados *parâmetros atuais* ou *atuais*, podem ser passados a um procedimento chamado, substituindo os parâmetros formais nesse corpo do procedimento. Os métodos para estabelecer a correspo-

```
 1) program sort(input, output);
 2) var a : array[0..10] of integer;
 3) procedure readarray;
 4) var i : integer;
 5) begin
 6)   for i := 1 to 9 do read(a[i])
 7) end;
 8) procedure quicksort(m, n : integer);
 9) var i, j, v, x : integer;
10) begin
11)   if n > m then begin
12)     i := m-1; j := n; v := a[n];
13)     while i+1 < j do begin
14)       x := a[i+1]; a[i+1] := a[j]; a[j] := x;
15)       if x < v then i := i+1
16)       else begin a[j] := a[i+1]; a[i+1] := x; j := j-1 end
17)     end;
18)     a[i] := a[m]; a[m] := a[j]; a[j] := v;
19)     quicksort(m, i-1); quicksort(i+1, n)
20)   end
21) begin
22)   readarray;
23)   a[0] := -9999; a[10] := 9999;
24)   quicksort(1, 9)
25) end.
```

Fig. 7.1. Um programa Pascal para ler e classificar inteiros

nos [illegible] parâmetros atuais e formais são discutidos na Seção 7.5. A linha 18 da Fig. 7.1 é uma chamada de [illegible] com parâmetros atuais [illegible].

Árvores de Ativações

Façamos as seguintes suposições sobre o fluxo de controle entre procedimentos durante a execução de um programa:

1. O controle sempre inicia no início do programa principal. A execução de um programa consiste em uma sequência de passos, com o controle, a cada passo, em algum procedimento específico do programa.
2. Cada execução de um procedimento começa no início do corpo do procedimento e eventualmente retorna imediatamente para o ponto imediatamente seguinte ao local de onde foi chamado. Isso significa que o fluxo de controle entre os procedimentos pode ser descrito usando-se árvores, como veremos em breve.

Cada execução do corpo de um procedimento é referenciada como uma ativação do procedimento. O tempo de vida de uma ativação de um procedimento p é a sequência de passos entre o primeiro e último passo na execução do corpo de p, incluindo o tempo despendido executando os procedimentos chamados por p, por procedimentos chamados por esses últimos e assim por diante. Informalmente, o termo "tempo de vida" se refere a uma sequência de passos consecutivos durante a execução de um programa.

Em linguagens como Pascal, cada vez que o controle entra num procedimento p, o procedimento de um procedimento q retornará eventualmente ao procedimento p. Uma ausência de um erro fatal. Mais precisamente, cada vez que o controle flui de uma ativação de um procedimento p para uma ativação de um procedimento q, o controle retornará para a mesma ativação de p.

Se as ativações de procedimentos não [illegible] os tempos de vida das ativações são sobrepostos ou são aninhados. Isto é, se a ativação p é iniciada antes de q ser deixada, o controle terá que abandonar q antes de deixar p.

Esta propriedade de aninhamento dos tempos de vida das ativações pode ser ilustrada inserindo-se dois comandos de impressão em cada procedimento, um antes do primeiro e outro depois do corpo do procedimento. O primeiro imprime uma [illegible] seguida pelo valor atual do procedimento e dos valores dos parâmetros atuais, e o último imprime [illegible] seguida da mesma informação. Uma execução do programa na Fig. 7.1 com esses comandos de impressão produz a saída mostrada na Fig. 7.2. O tempo de vida da ativação [illegible] é a sequência de passos executados entre a impressão de [illegible]. Na Fig. 7.2, é assumido que o valor retornado por [illegible].

Um procedimento recursivo pode ter muitas ativações ativas ao mesmo tempo. A Fig. 7.3 mostra que o controle entra na ativação de [illegible] a partir da linha 24 no início da execução do programa, mas abandona essa ativação quase no fim. Nesse meio tempo, existem duas outras ativações de [illegible] e, dessa forma, [illegible] é recursivo.

Um procedimento recursivo não precisa chamar a si mesmo diretamente; p pode chamar um outro procedimento q que pode chamar p através de alguma sequência de chamadas de procedimentos. Podemos usar uma árvore, chamada de árvore de ativações, para descrever como o controle entra e abandona as ativações. Numa árvore de ativações:

1. cada nó representa uma ativação de um procedimento;
2. a raiz representa a ativação do programa principal;
3. o nó a está acima do nó b se e somente se o controle flui de a para b;

Fig. 7.2. Saída supondo-se ativações de procedimentos na Fig. 7.1.

4. o nó a está à esquerda do nó b se e somente se o tempo de vida de a ocorre antes do tempo de vida de b.

Como cada nó representa uma única ativação e vice-versa, é conveniente falarmos do controle estando em um nó quando, estritamente falando, ele está sendo representado por aquele nó.

Exemplo 7.1. A árvore de ativações na Fig. 7.3 é construída a partir da saída na Fig. 7.2. Para começar, o programa principal [illegible] é representado pelo nó mais à esquerda. A ativação [illegible] é representada pelo segundo filho da raiz, e [illegible] com parâmetros atuais 1 e 9. Durante essa ativação, as chamadas de [illegible] nas linhas 16 e 18 da Fig. 7.1 levam às ativações [illegible]. Note-se que as ativações [illegible] são recursivas e que começam e terminam antes de [illegible] terminar. □

Pilhas de Controle

O fluxo de controle num programa corresponde a uma travessia em profundidade da árvore de ativações que começa à raiz, visita um nó antes de seus filhos e recursivamente visita os filhos à esquerda num nó em ordem da esquerda para a direita. A saída da Fig. 7.2 ordena, por consequência, as ativações de uma travessia da árvore de ativações na Fig. 7.3. Imprimindo a cadeia [illegible] quando o nó de uma ativação for alcançado pela primeira vez e imprimindo a cadeia [illegible] após toda a subárvore do nó ter sido visitada durante a travessia.

Podemos usar uma pilha, chamada de pilha de controle, para controlar a vida das ativações dos procedimentos. A ideia é empilhar o nó para uma ativação na pilha de controle quando a ativação é iniciada e remover o nó quando a ativação termina. Dessa forma, o conteúdo da pilha de controle está relacionado a percursos até a raiz da árvore de ativações. Quando um nó está no topo da pilha de controle, a pilha contém os nós ao longo do percurso de a até a raiz.

Exemplo 7.2. A Fig. 7.4 ilustra os dois últimos de uma série de situações que foram exibidas quando o controle entrou na ativação representada por [illegible]. As ativações com nós [illegible] já haviam terminado e foram [illegible] executadas até o fim e, consequentemente, a figura contém apenas [illegible] para seus nós. As linhas sólidas [illegible] percorrem [illegible] até a raiz.

A cada momento, a pilha de controle contém os nós ao longo desse percurso até a raiz. O topo da pilha está à direita.

[illegible] Os chamados [illegible] podem [illegible]. Veja [illegible] Hopcroft e Ullman [1979] para detalhes sobre [illegible]. A Fig. 7.3 representa a [illegible] árvore de ativações [illegible] Fig. 7.2 [illegible] [illegible] [illegible]

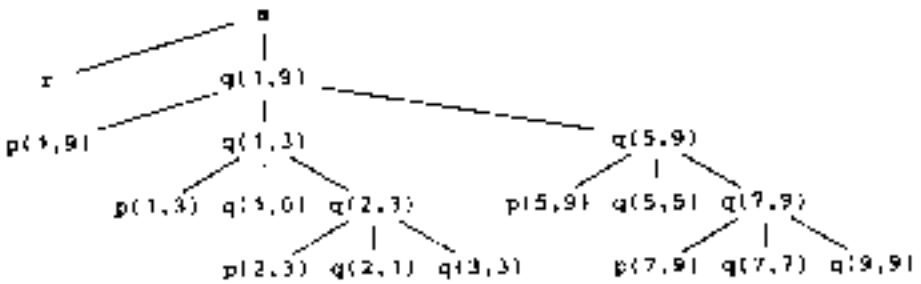


Fig. 7.4 Uma árvore de ativação correspondente [illegible]

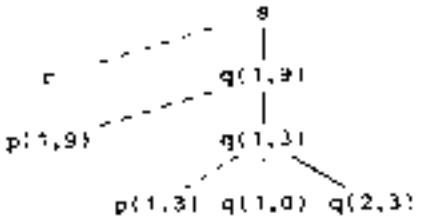


Fig. 7.5. [illegible]

[illegible]

[illegible]

Uma das pilhas de controle mais importantes é a pilha de ativações de procedimentos, usada para implementar linguagens como Pascal e C. Esta técnica é descrita em detalhes nas Seções 7.3 e 7.4.

O Escopo de uma Declaração

Uma declaração numa linguagem é uma construção sintática que associa informações a um nome. Tais declarações podem ser explícitas, como na linguagem Pascal

[illegible]

ou podem ser implícitas. Por exemplo, num programa Fortran, qualquer ocorrência de um identificador não declarado pertencente a uma atribuição declara implicitamente que seja declarado explicitamente algo diferente. As regras de escopo de uma linguagem determinam como a ocorrência de um nome se aplica a uma declaração particular num programa fonte. No programa Pascal da Fig. 7.1, o identificador i é declarado três vezes, nas linhas 4, 9 e 13, e cada declaração de nome i pertence a um procedimento diferente. Essas três declarações de i são independentes umas das outras. A declaração na linha 4 se aplica a todas as ocorrências de i na linha 5 e também a todas as ocorrências de i na linha 8 que estão fora do escopo da declaração na linha 13. As duas ocorrências de i na linha 13 estão no escopo da declaração na linha 13.

A parte do programa a que uma declaração se aplica é chamada de escopo daquela declaração. Uma ocorrência de um nome local num procedimento p pode estar num escopo estático ou dinâmico. A discussão sobre escopo estático e dinâmico nesta seção se concentra nos nomes locais a procedimentos e supõe que o escopo estático de uma declaração local seja sempre igual ao seu escopo dinâmico. A distinção entre nomes locais e não locais vem a ser mais importante para construções mais complexas que possam ter declarações aninhadas.

Como o escopo de uma declaração é uma propriedade da declaração de um nome, e ainda vezes conveniente usar a abreviação "escopo de um nome" em lugar de "escopo da declaração do nome que se aplica a esta ocorrência de nome". Neste sentido, o escopo [illegible]

[illegible]

[illegible]

Em tempo de compilação, a tabela de símbolos pode ser usada para encontrar a declaração que se aplica a uma ocorrência de um nome. Quando a declaração é encontrada, é criada para o nome uma entrada na tabela de símbolos. Nas linguagens que estamos considerando, uma declaração é reconhecida quando a sintaxe for processada. As tabelas de símbolos são discutidas na Seção 7.6.

Amarrações de Nomes

Mesmo que um programa seja declarado uma única vez, o mesmo pode ser executado para definir objetos de dados diferentes em tempos de execução. O tempo entre quando "objeto de dados" corresponde a uma localização de memória que pode abrigar valores.

Na semântica das linguagens de programação, associamos variáveis e seus valores através de uma função que mapeia uma localização de memória (o valor-l da variável) [illegible] Fig. 7.6. Usamos os termos valores-l e valores-r, provenientes do Capítulo 2, simplesmente para nos referir a um valor l ou um valor r.

Fig. 7.6. [illegible]

Os ambientes e estados são diferentes, uma atribuição modifica o estado, mas não o ambiente. Por exemplo, suponha que o endereço de memória [illegible] associado à variável [illegible] abrigue o valor 0. Após a atribuição [illegible] o mesmo endereço de memória [illegible] está associado a [illegible] mas o estado armazenado [illegible]

Quando um ambiente associa uma localização de memória a um nome, dizemos que o nome está armazenado nessa localização. É comum referenciada como uma associação que [illegible] O termo "localização" de memória é tratado internamente. Se o nome corresponde a um tipo básico, o armazenamento para ele pode ser uma coleção de palavras de memória.

Uma amarração é a contraparte dinâmica de uma declaração, como mostrado na Fig. 7.6. Como temos visto, pode haver mais de uma amarração de um mesmo nome que pode estar ativa ao mesmo tempo. Em Pascal, por [illegible] de escopo estático, a um procedimento recursivo estão amarrados a diferentes localizações de memória a cada ativação do mesmo. As [illegible] sobre amarrações de nomes locais são [illegible] na Seção 7.5.

Noção Estática	Contraparte Dinâmica
definição de um procedimento	ativação de um procedimento
declaração de um nome	amarrações de um nome
escopo de uma declaração	tempo de vida de uma amarração

Fig. 7.6. [illegible]

Perguntas a Responder

A forma pela qual um compilador de uma linguagem precisa organizar sua memória em tempo de execução é amplamente determinada por respostas a questões como as seguintes:

1. Os procedimentos podem ser recursivos?
2. O que acontece com valores de nomes locais quando o controle retorna da execução de um procedimento?
3. Um procedimento pode se referir a nomes não-locais?
4. Como são transmitidos os parâmetros quando um procedimento é chamado?
5. Os procedimentos podem ser transmitidos como parâmetros?
6. Os procedimentos podem ser retornados como resultados?
7. A memória pode ser alocada dinamicamente sob controle do programa?
8. A memória precisa ser liberada explicitamente?

O efeito desses tempos no espaço necessário em tempo de execução para uma dada linguagem de programação é examinado no restante deste capítulo.

7.2 ORGANIZAÇÃO DE MEMÓRIA

A organização de memória em tempo de execução discutida nesta seção pode ser usada em linguagens tais como Fortran, Pascal e C.

Subdivisão da Memória em Tempo de Execução

Suponhamos que o compilador obtenha um bloco de memória para o sistema operacional para o programa compilado executar no momento. A partir da discussão da última seção, essa memória em tempo de execução poderia ser subdividida, de forma a abrigar:

1. o código-alvo gerado,
2. objetos de dados, e
3. uma contagem da pilha de controle para controlar ativações de procedimentos.

O tamanho do código gerado é estabelecido em tempo de compilação e dessa forma o compilador pode colocá-lo em uma área estaticamente determinada. Talvez na parte mais baixa da memória. Analogamente, os tamanhos de alguns objetos de dados também já devem ser conhecidos em tempo de compilação, e esses objetos de dados também podem ser colocados numa área estaticamente determinada, como na Fig. 7.7. Uma razão para a alocação estática de tantos objetos de dados quanto o possível é que os endereços desses objetos de dados podem ser compilados no código alvo. Todos os objetos de dados em Fortran podem ser alocados estaticamente.

As implementações de linguagens como Pascal e C usam pilhas de controle para gerenciar as ativações dos procedimentos. Quando ocorre uma chamada, a execução de uma ativação é interrompida e as informações sobre o estado da máquina, tal como o valor do apontador da pilha e uma instrução e dos registradores de máquina, são salvas na pilha. Quando o controle retorna da chamada, essa ativação pode ser retomada após a restauração dos valores dos registradores salvos e o estabelecimento do apontador da pilha para a próxima instrução para um ponto imediatamente após a chamada. Os objetos de dados cujos tempos de vida estão contidos naquela ativação podem ser estabelecidos na pilha, juntamente com outras informações associadas à ativação. Essa estratégia é discutida na próxima seção.

Uma área separada de memória em tempo de execução, chamada de *heap*, abriga todas as outras informações. Pascal permite que os dados sejam alocados sob controle do programa, como discutido na Seção 7.4, e o *heap* é o armazenamento para tais dados obtidos a partir do *heap*. As implementações de linguagens nas quais os tempos de vida das ativações não possam ser representados por uma árvore de ativações podem usar o *heap* para manter informações sobre as ativações. A razão complexa pela qual os dados são preservados é mantida numa pilha como nas duas colocações numa pilha em vez de num *heap*.

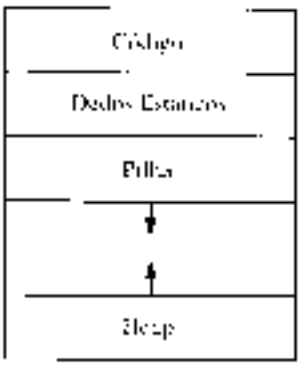


Fig. 7.7. Subdivisão típica da memória em tempo de execução em áreas de código, dados estáticos, pilha e *heap*.

Os tamanhos da pilha e do *heap* podem variar à medida que o programa executa e, a fim de maximizar a utilização do espaço, eles são colocados como na Fig. 7.7, onde podem crescer em direções um do outro, conforme necessário. Pascal e C precisam apenas de uma pilha porque exceções quanto ao uso de ponteiros nessas linguagens precisam ser tratadas.

Por convenção, a pilha cresce para baixo (isto é, "para o topo" da página). Como os endereços de memória aumentam à medida que vamos para baixo na página, "crescimento para baixo" significa em direção aos endereços maiores. Se o topo da pilha, os deslocamentos a partir do topo da pilha podem ser computados, subtraindo-se os deslocamentos de zero. Em muitas máquinas, esse cômputo pode ser feito enquanto um valor de topo é mantido em registrador. Os endereços da pilha podem então ser representados como deslocamentos a partir de topo.

Registros de Ativação

As informações necessitadas por uma única execução de um procedimento são gerenciadas utilizando-se um bloco contíguo de memória conhecido como chamado de um *registro de ativação* ou *quadro*, consistindo em uma coleção de campos mostrados na Fig. 7.8. Nem todas as linguagens, nem todos os compiladores, usam todos esses campos frequentemente, os registradores podem tomar o lugar de um ou mais deles. Para linguagens como Pascal e C, o usual é empilhar o registro de ativação do procedimento na pilha em tempo de execução quando o procedimento é chamado e remover a ativação quando o controle retorna ao chamador.

O propósito dos campos de um registro de ativação é como segue, começando com o campo para dados temporários.

1. Valores temporários, tais como aqueles que emergem da avaliação de expressões, são armazenados no campo para temporários.
2. O campo para dados locais abriga aqueles dados que são locais a uma execução do procedimento. A disposição desse campo é discutida em seguida.

Digressão: Na Fig. 7.7 o espaço é mantido entre as áreas de pilha e *heap* para crescimento. Na verdade, a pilha e o *heap* crescem um em direção ao outro, se necessário, em sistemas de tempo de execução que suportam explicitamente ambos os mecanismos. Quando a pilha e o *heap* crescem um em direção ao outro, o espaço é usado de forma mais eficiente, mas a implementação é mais difícil. Por exemplo, quando um bloco de memória é liberado pelo programa, o sistema de tempo de execução pode ter que mover blocos para evitar que a memória livre se torne fragmentada em muitos blocos pequenos.

j) O campo para o estado salvo da máquina deve conter informações sobre o estado da máquina exatamente antes do procedimento ser chamado. Essa informação inclui os valores de um conjunto de registradores de máquina que precisam ser restaurados quando ocorre o retorno do procedimento.

k) O campo opcional descrito na Seção 7.4 para retornar os valores alocados em registros de ativação. Esses valores de acesso são necessários para uma linguagem como Fortran, porque esses locais são mantidos em localizações fixas. Os valores de acesso são inexistentes em linguagens como Pascal.

l) O campo opcional aponta para o registro de ativação do chamador.

m) O campo para os parâmetros atuais será usado pelo procedimento chamador para fornecer os parâmetros do procedimento chamado. Mas não há espaço aqui, os parâmetros do registro de ativação, mas, na prática, os mesmos são passados em registradores de máquina, para uma eficiência maior.

n) O campo para o valor retornado será usado pelo procedimento chamado para fornecer um valor para o procedimento chamador. De novo, na prática esse valor é frequentemente retornado num registrador, por razões de eficiência.

Qualquer um desses campos pode ser determinado em tempo de chamada do procedimento. De fato, os tamanhos de quase todos os campos podem ser determinados em tempo de compilação. Uma exceção óbvia é quando um procedimento pode ter um número variável de parâmetros, sendo cada um deles especificado pelo valor de um parâmetro atual disponível somente quando o procedimento for chamado em tempo de execução. Veja a Seção 7.3 para o estabelecimento de dados de tamanho variável em registros de ativação.

Gabarito para os Dados Locais em Tempo de Compilação

Suponhamos que o armazenamento em tempo de execução venha em blocos de bytes contíguos, onde a *byte* é a menor unidade endereçável de memória. Em muitas máquinas, um *byte* é constituído por oito bits e alguns números consecutivos de bytes formam uma palavra de máquina. Múltiplos objetos são armazenados em bytes consecutivos, sendo o endereço do primeiro byte de cada sequência.

A quantidade de memória necessária por um nome é determinada pelo seu tipo. Um tipo de dados elementar, tal como caractere, inteiro ou real, pode ser usualmente armazenado num número inteiro de bytes. A memória de armazenamento de um agregado, como um array ou registro, precisa ser suficientemente ampla para abrigar todos os seus componentes. Para facilidade de acesso aos componentes, o armazenamento para os agregados é frequentemente alocado em blocos de bytes contíguos. Veja as Seções 8.2 e 8.3 para mais detalhes.

O cálculo para dados locais é estabelecido na medida em que declarações de um procedimento são examinadas em tempo de compilação. Dados de tamanho variável são mantidos fora desse campo. Mantemos um contador de localizações de memória que tenham sido reservadas para as declarações anteriores. A partir do contador, determinamos o endereço relativo de armazenamento para um objeto de dados local em relação a alguma posição, tal como o início do registro de ativação. O endereço relativo de um dado armazenado é a diferença entre os endereços da posição e do objeto de dados.

A disposição da memória para os objetos de dados é fortemente influenciada pelas restrições de endereçamento da máquina-alvo. Por exemplo, as instruções para acessar inteiros podem esperar que os inteiros sejam *alinhados*, isto é, colocados em certas posições na memória, tal como num endereço divisível por 4. Apesar de um vetor de dez caracteres precisar somente de dez bytes para abrigá-los, um compilador pode, portanto, reservar 12 bytes, deixando 2 bytes sem uso. O espaço deixado sem uso devido às considerações de alinhamento é denominado de *buraco*. Quando o espaço estiver a preços, um compilador pode reorganizar dados de forma que nenhum buraco seja gerado, pode haver necessidade de instruções adicionais em tempo de execução para posicionar os dados compactados de forma que possam ser operados como se estivessem propriamente alinhados.

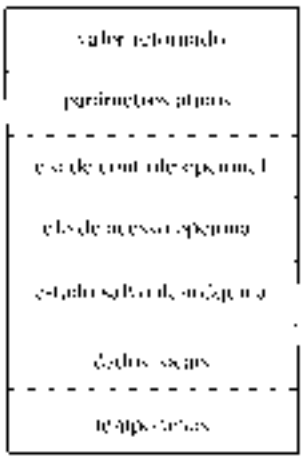


Fig. 7.8. Um registro de ativação típico.

Exemplo 7.3. A Fig. 7.9 é uma simplificação da disposição de dados usada em compiladores C para duas máquinas que chamamos Máquina 1 e Máquina 2. Os procedimentos manipulam três tamanhos de inteiros, declarados usando as palavras-chave [illegible]. Os comprimentos de inteiros para as duas máquinas são tais que o compilador para a Máquina 1 reserva 16, 32 e 32 *bits* para os três tamanhos de inteiros, enquanto que o compilador para a Máquina 2 reserva 24, 48 e 64 *bits*, respectivamente. Para uma comparação entre as máquinas, os comprimentos são medidos em *bits* na Fig. 7.9, ainda que nenhuma das máquinas permita que *bits* sejam endereçados diretamente.

A memória da Máquina 1 é organizada em *bytes* consistindo em 8 *bits* cada um. Ainda que cada byte possua um endereço, o compilador de instruções favorece que os inteiros curtos sejam providenciados em endereços divisíveis por 4. O compilador coloca os inteiros curtos em endereços pares, mesmo que no processo tenha que pular um byte de enchimento. Por conseguinte, quatro *bytes*, consistindo em 32 *bits*, podem ser reservados para um caractere seguido por um inteiro curto.

Na Máquina 2 cada palavra consiste em 64 *bits* e são permitidos 24 *bits* para o endereçamento de uma palavra. Existem 64 possibilidades para os *bits* individuais dentro de uma palavra, de forma que 6 *bits* adicionais são necessários para distinguir-los. Por projeto, um apontador para um caractere na Máquina 2 gasta 30 *bits* — 24 para encontrar a palavra e 6 para a posição do bit inicial do caractere dentro da palavra.

A forte orientação para palavra dos conjuntos de instruções da Máquina 2 levou o compilador a reservar uma palavra completa de cada vez, mesmo que um número menor de *bits* fosse suficiente para representar todos os possíveis valores de um tipo. Por exemplo, somente 8 *bits* são necessários para representar um caractere. Por conseguinte, sob o alinhamento, a Fig. 7.9 mostra os 64 *bits* para cada tipo. Dentro de cada palavra, os *bits* para cada tipo são colocados nas posições especificadas. Duas palavras consistindo em 128 *bits* seriam reservadas para um caractere seguido por um inteiro curto, com os caractere usando apenas 8 *bits* da primeira palavra e o inteiro curto somente 24 *bits* da segunda. ■

---

1 Existem algumas máquinas que não endereçam bytes, e algumas que não endereçam nada menor que uma palavra.

Tipo	Tamanho (bits)		Alinhamento (bits)	
	Máquina 1	Máquina 2	Máquina 1	Máquina 2
	8	8	8	64
	16	21	16	64
	32	48	32	64
	32	64	32	64
	32	64	32	64
	64	128	32	64
apontador de caracteres	32	80	32	64
outros apontadores	32	24	32	64
estruturas	8	64	32	64

Fig. 7.9. Dispositivos de dados usados por um compilador.

7.3 ESTRATÉGIAS PARA A ALOCAÇÃO DE MEMÓRIA

Uma estratégia diferente de reserva de memória é usada em cada uma das três áreas de dados na organização da Fig. 7.8:

1. A alocação de memória estática designa armazenamento para todos os objetos de dados em tempo de compilação.
2. A alocação de memória de pilha gerencia a memória em tempo de execução como uma pilha.
3. A alocação de memória heap reserva e libera áreas de armazenamento em uma ordem de acesso de dados qualquer em tempo de execução, a partir de uma área de dados sempre crescente.

Essas estratégias de alocação são aplicadas, respectivamente, aos três tipos de ativação. Também descrevemos como o código-alvo de um procedimento tem acesso à memória alocada a um nome local.

Alocação de Memória Estática

Na alocação estática, os nomes são amarrados a localizações de memória à medida que o programa é compilado, de forma que não há necessidade de um pacote de suporte em tempo de execução. Uma vez que as amarrações não se modificam em tempo de execução, cada vez que um procedimento é ativado, seus nomes são amarrados às mesmas localizações de memória. Essa propriedade permite que os valores dos nomes locais sejam retidos através das ativações de um procedimento. Isto é, quando o controle retorna a um procedimento, os valores dos objetos de dados locais são os mesmos que detinham quando o controle deixou-o pela última vez.

A partir do tipo de um nome, o compilador determina a quantidade de memória a ser reservada para o mesmo, como descrito na Seção 7.1. O endereço dessa memória consiste no deslocamento, a partir de uma das extremidades do registro de ativação do procedimento, do compilador ter a escolha de decidir onde os registros de ativação devem ficar, também, qual a posição relativa no código-alvo do programa dos nomes em suas posições relativas entre si. Uma vez que essa decisão tenha sido tomada, a posição de cada registro de ativação e, consequentemente, a de armazenamento de cada nome nos registros são fixadas. Em tempo de compilação, podemos, portanto, preencher os endereços nos quais o código-alvo precisa encontrar os dados sobre os quais opera. Similarmente, os endereços nos quais as informações devem ser salvas quando ocorre uma chamada de procedimento também são conhecidos em tempo de compilação.

No entanto, algumas limitações de compilação à alocação de memória estática aparecem imediatamente a mesma é utilizada:

1. O tamanho dos objetos de dados e as restrições a respeito de suas posições na memória podem ser conhecidos em tempo de compilação.
2. Os procedimentos recursivos são proibidos, porque todas as ativações de um procedimento usam as mesmas amarrações para nomes locais.
3. As estruturas de dados não podem ser criadas dinamicamente, uma vez que não há mecanismo para controle de memória em tempo de execução.

Formam-se pesquisas para possibilitar a reserva de memória estática em programas baseados em um programa principal de sub-rotinas e funções e também as estruturas de controle, como, por exemplo, a Fig. 7.10. Usando a organização de memória da Fig. 7.9, a disposição do código e dos registros de ativação para esse programa é mostrada na Fig. 7.11. Dentro do registro de ativação para [illegible] uma "constante" — [illegible] espaço para os objetos de dados locais [illegible] A memória amarrada a [illegible] abriga uma cadeia de 80 caracteres. É seguida por espaço para guardar um valor numérico para [illegible] e um valor de caractere para [illegible] O fato de [illegible] ser também declarado em [illegible] não apresenta problema, porque os objetos de dados locais para os dois procedimentos obtêm espaço nos seus respectivos registros de ativação.

Como os amarrados do código executável do registro de ativação são conhecidos em tempo de compilação, são possíveis organiza-

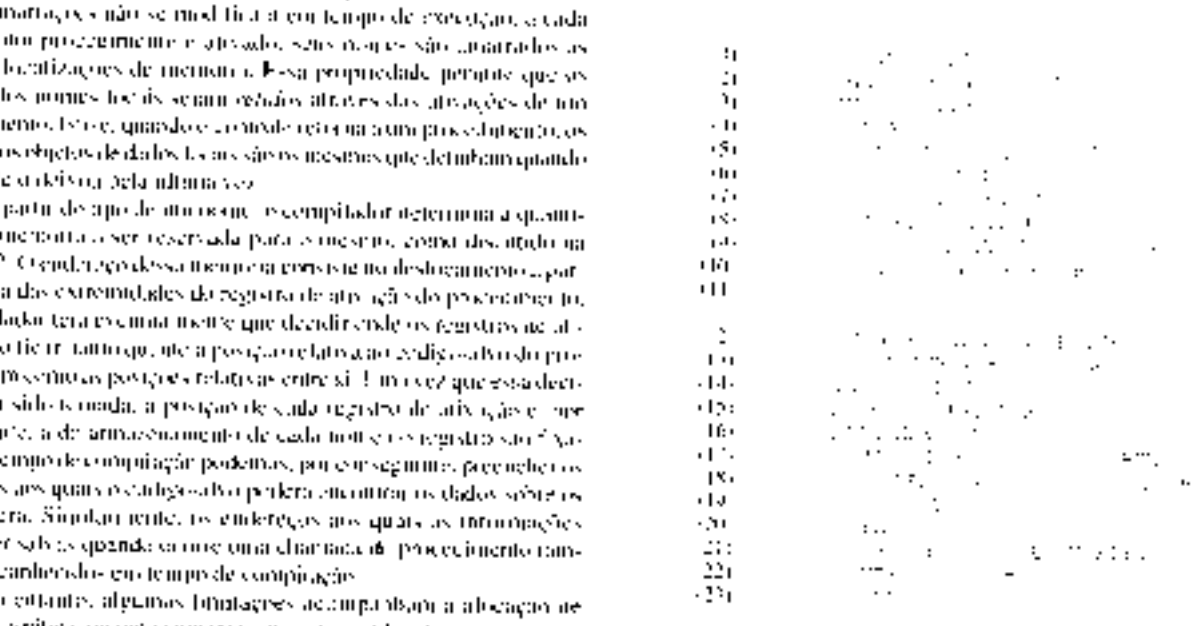


Fig. 7.10. Um programa [illegible]

[illegible]

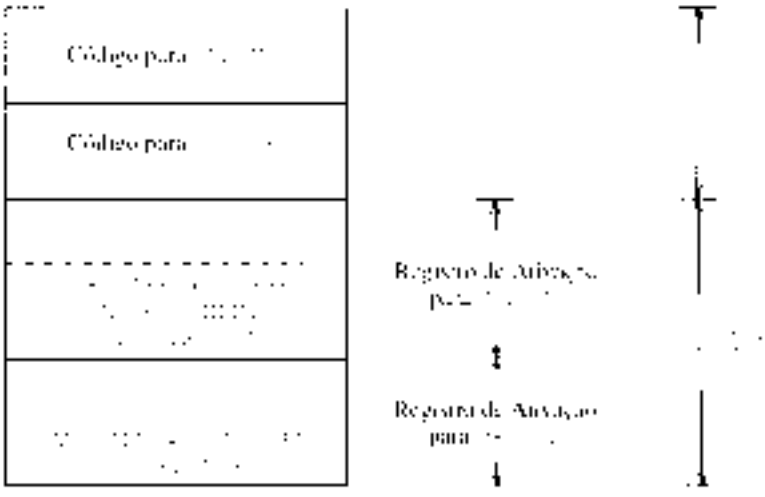


Fig. 7.11. Armazenamento estático para os objetos de dados locais de cada procedimento, como ilustrado na Figura 7.7.

cável memória que nas aquelas [illegible] Fig. 7.11. [illegible] compilador [illegible] pode [illegible] o registro de ativação para um procedimento, [illegible] como código desse procedimento. Em alguns sistemas de computação [illegible] a facilidade de sobreposição relativa dos registros de ativação correspondentes, [illegible] permite que o editor de ligações ligue os registros de ativação [illegible] valor executável.

Exemplo 7.4. O programa [illegible] Fig. 7.11 [illegible] com que os valores [illegible] objetos de dados locais sejam retidos ao longo das ativações [illegible] procedimentos. [illegible] Fig. 7.7 especifica que o valor de um objeto de dados local [illegible] ativação pode ser o mesmo que antes da última ativação. Os valores iniciais para esses objetos de dados locais podem ser especificados [illegible] uma vez.

Se [illegible] a linha 19 do procedimento [illegible], [illegible] acesso de cada vez para *before*. O procedimento delivera caracteres sucessivos a cada vez que for ativado. O programa principal [illegible] também possui um *buffer* no qual acumula caracteres até que um espaço seja encontrado. A entrada

[illegible]

os caracteres retornados pelas ativações de [illegible] Cat declarados na Fig. 7.12 a saída do programa é

[illegible]

O objetivo principal [illegible] facilitar precisamente essa [illegible] entre ativações. O comando [illegible] a linha 15 assegura que, quando o controle retornar [illegible] os objetos de dados locais associados a [illegible] conterão os valores que tinham quando o controle deixou o procedimento pela última vez. Na primeira vez que o controle [illegible] o valor do objeto de dados local associado [illegible] é obtido a partir do valor especificado no enunciado [illegible] linha 16. Dessa forma, [illegible] é inicializado com 81.

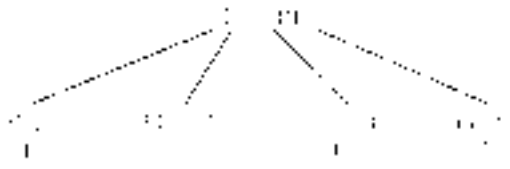


Fig. 7.12. Caracteres retornados pelas ativações de [illegible].

Alocação de Memória de Pilha

A alocação de memória de pilha está baseada [illegible] uma pilha de controle. A sequência organizada que [illegible] pilha [illegible] gestão de ativações empilhados e desempilhados [illegible] ativações correspondentes, respectivamente. O armazenamento para objetos de dados locais [illegible] de um procedimento [illegible] ativação [illegible] Dessa forma, objetos de dados locais estão associados a uma ativação [illegible] cada ativação porque um novo registro de ativação é empilhado quando uma chamada é realizada. Sobre o valor [illegible] objetos de dados locais [illegible] ativação [illegible] objetos de dados locais [illegible] ativação [illegible] porque [illegible] para os mesmos desaparecem quando o registro de ativação é desempilhado.

Primeiro descrevemos a forma para uma alocação de memória de pilha, na qual os tamanhos de todos os registros de ativação são [illegible] conhecidos [illegible] compilação. As situações [illegible] quando [illegible] informações incompletas a respeito dos tamanhos em tempo de compilação [illegible] são [illegible].

Suponha [illegible] que o registro de ativação [illegible] topo da pilha. Em tempo de execução, um registro de ativação [illegible] ser alocado e liberado [illegible] chamando [illegible] e [illegible], respectivamente, [illegible] procedimentos [illegible] registro de ativação [illegible] um [illegible] [illegible] [illegible] [illegible] [illegible] código do [illegible] ser executado. Quando o controle retorna [illegible] [illegible] [illegible].

Exemplo 7.5. A Fig. 7.13 mostra os registros de ativação que são empilhados e desempilhados em tempo de execução a medida que o controle flui através [illegible] árvore de ativações da Fig. 7.3. As linhas pontilhadas na árvore [illegible] para ativações que [illegible]. A [illegible] corresponde [illegible] ativação do procedimento [illegible]. Quando o controle atinge a primeira chamada [illegible], o procedimento [illegible] ativados seu registro de ativação [illegible] empilhados. Quando o controle retorna dessa ativação, o registro é desempilhado, deixando somente o registro para [illegible] pilha. Na ativação de [illegible] o controle atinge a chamada de [illegible], [illegible] um registro de ativação para a ativação de [illegible] alocado no topo da pilha. Sempre que o controle estiver numa ativação, seu registro de ativação estará no topo da pilha.

Várias ativações [illegible] entre dois instantâneos da Fig. 7.13. No último instantâneo, as ativações [illegible] terminaram durante o tempo de vida de [illegible] e, dessa forma, seus registros de ativação [illegible] se [illegible] da pilha, deixando o registro de ativação para [illegible] no topo.

Num procedimento Pascal, [illegible] é [illegible] um [illegible] para [illegible] locais [illegible] registro de ativação, como descrito na Seção 7.2.

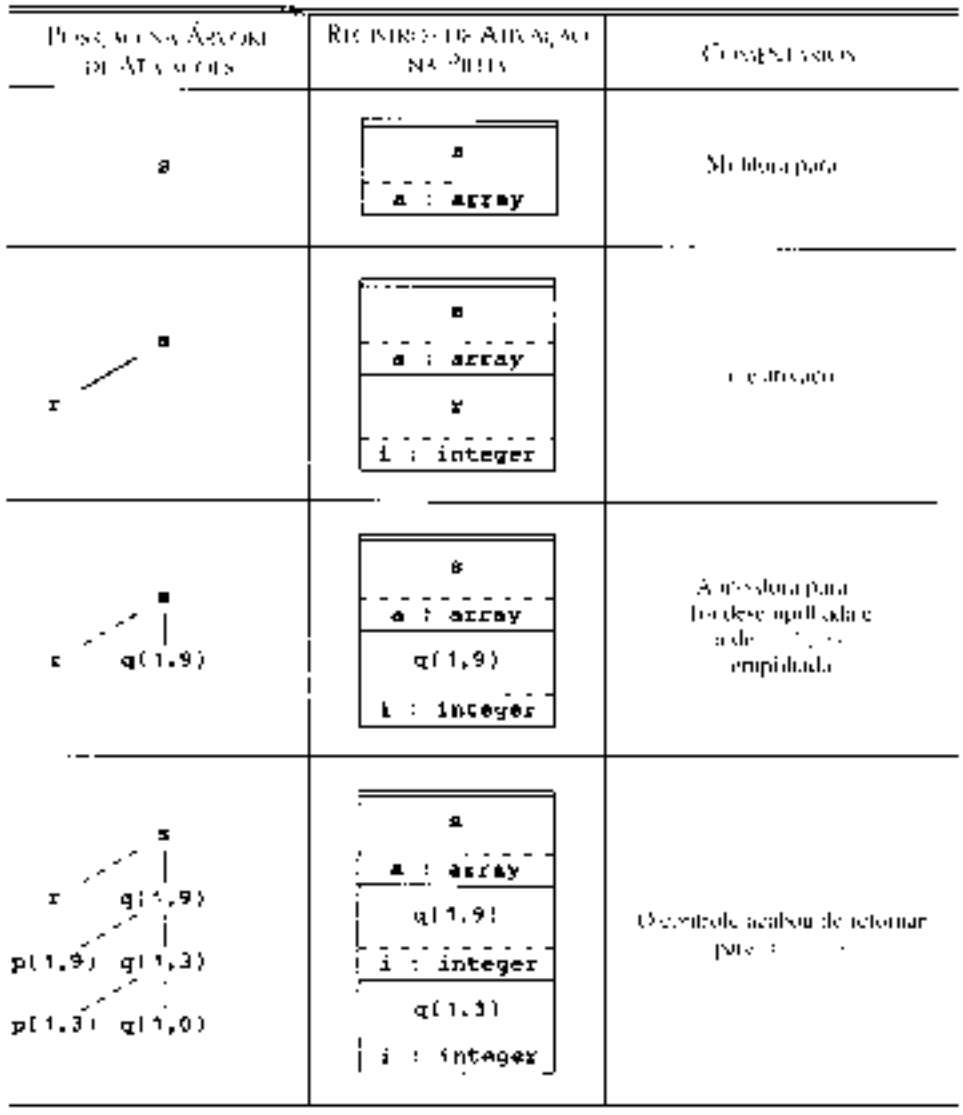


Fig. 7.13. Alocação de registros de ativação numa pilha de execução

em tempo de execução, supõe-se que seja possível localizar a localização de um de um registro. O endereço para um nome local num código-objeto para procedimentos poderia, consequentemente, ser escrito como deslocamento, para um registro de ativação que pode ser encontrado pela adição de um valor do registrador-base. Deste modo as referências podem ser alternativamente vistas como deslocamentos a partir do valor de qualquer outro registrador apontando para uma localização dentro dos registros de ativação.

Sequências de Chamada

As chamadas de procedimento são implementadas através da geração de uma *sequência de chamada* e de uma *sequência de retorno* no código-objeto. Uma sequência de chamada aloca um registro de ativação e preenche as informações em seus campos. Uma sequência de retorno restaura o estado da máquina de tal forma que o procedimento chamador possa continuar sua execução.

As sequências de chamada e os registros de ativação diferem muito para as implementações de uma mesma linguagem. O código numa sequência de chamada é frequentemente dividido entre o procedimento chamador (o chamador) e o procedimento chamado (o procedimento chamado). Não existe uma divisão exata das tarefas em tempo de execução entre o procedimento chamador e o chamado — a linguagem-fonte, a máquina alvo e o sistema operacional impõem exigências que podem favorecer uma solução ou outra.[1]

Um princípio que orienta o projeto das sequências de chamada e dos registros de ativação é que os campos cujos tamanhos são conhecidos mais cedo sejam colocados no meio. No modelo geral de um registro de ativação da Fig. 7.8, o elo de controle, o de acesso e os campos de estado da máquina ficam no meio. A decisão a respeito de onde colocar os elos de controle e de acesso é parte do projeto do compilador, de forma que esses campos podem ser livados em tempo de construção do compilador. Se exatamente a mesma quantidade de informações de estado da máquina for guardada para cada ativação, o mesmo código pode realizar o armazenamento e restauração para todas as ativações. Sobretudo, programas tais como depuradores terão uma vida facilitada ao localizar o conteúdo da pilha quando ocorrer um erro.

Ainda que o tamanho do campo para os objetos de dados temporários seja conhecido, o alocador pode colocá-los em tempo de compilação, de modo que o acesso a eles também seja indireto através do registrador-base.

[1] Algumas máquinas têm instruções especiais de chamada e retorno que podem ser usadas, mas estas instruções frequentemente não são suficientemente flexíveis para as necessidades de todas as implementações de uma linguagem. Em vez disso, elas são usadas apenas por implementações simples que não requerem muito suporte em tempo de execução.

também pode não ser exatamente o que a máquina de computador. Uma interpretação de código ou de armazenamento pode reservar um número de objetos de dados temporários necessários pelo procedimento enquanto ele está em execução, mas o compilador pode não conhecer esse número. Por conseguinte, o procedimento precisa de registros de ativação, e o espaço para esse campo pode ser alocado para dados locais após as alterações em seu tamanho. Também os deslocamentos relativos dos objetos de dados dos campos no novo registro de ativação

Uma vez que cada chamada possui os seus próprios parâmetros, é preciso que os valores atuais dos parâmetros sejam armazenados no registro de ativação do procedimento chamado. Os detalhes de transmissão de parâmetros são discutidos na Seção 7.5. Na rotina em tempo de execução, o registro de ativação do chamado é imediatamente abaixo daquele para o chamador, como na Fig. 7.11. Há uma vantagem em se acessar os campos para os parâmetros reais e o valor previamente armazenado no topo do registro de ativação do chamado. O chamador pode ter acesso a esses campos utilizando deslocamentos a partir do início de seu próprio registro de ativação, sem conhecer a disposição exata dos registros de ativação do procedimento chamado. Em particular, torna-se uma tarefa fácil para o chamador saber a respeito de dados locais apropriados do procedimento chamado. Um benefício dessa disposição de informações está em que os procedimentos com um número variável de argumentos, tais como em C, podem ser tratados, como discutido abaixo.

Em linguagens como Pascal em que os dados locais de procedimentos podem ter um tamanho que possa ser determinado em tempo de compilação, o mais frequentemente o comprimento de um registro só pode depender do valor de um parâmetro transmitido a um procedimento. Nesse caso, o tamanho de todos os dados locais pode ser finalmente determinado assim que o procedimento seja chamado. As técnicas para o armazenamento de dados de tamanho variável são discutidas adiante nesta seção.

A sequência de chamada seguinte é produzida pela discussão acima. Como na Fig. 7.14, o registro de ativação para o final do campo de estado da máquina no registro de ativação. Esta posição é selecionada pelo chamador, de forma que o valor pode ser formado rapidamente, só pelo estabelecimento de ap_p imediatamente antes de saltar para o procedimento chamado. O código para o procedimento chamado pode acessar seus temporários e dados locais usando deslocamentos a partir de ap_p. A sequência de chamada é:

1. O chamador avalia seus parâmetros atuais.
2. O chamador armazena um endereço de retorno e o valor antigo de ap_p no registro de ativação do procedimento chamado. O chamador então altera ap_p para a posição mostrada na Fig. 7.14. Isto é, ap_p é movido para além dos dados locais, temporários do chamador e dos parâmetros e campos de estado do procedimento chamado.
3. O procedimento chamado salva os valores dos registradores e outras informações de estado.
4. O procedimento chamado inicializa seus dados locais e começa a execução.

Uma possível sequência de retorno é:

1. O procedimento chamado coloca um valor de retorno no registro de ativação do chamador.
2. Usando as informações no campo de estado da máquina, o procedimento chamado restaura ap_p e outros registradores e desvia para um endereço de retorno no código do procedimento chamador.
3. Apesar de ap_p ter sido decrementado, o procedimento chamador pode encontrar o valor retornado em seu próprio registro de ativação, usá-lo para avaliar uma expressão.

As sequências de chamada acima permitem que o número de argumentos do procedimento chamado dependa da chamada. Note-se que, em tempo de compilação, o código-alvo do chamador conhece o número de argumentos que está fornecendo ao procedimento chamado. Por conseguinte, o chamador conhece o tamanho do campo de parâmetros. Entretanto, o código-alvo do procedimento chamado precisa ser preparado para tratar outras chamadas igualmente válidas, possivelmente esperando ser chamado por outro, com um número de parâmetros. Usando-se a organização da Fig. 7.14, as instruções destinadas aos parâmetros precisam ser contíguas ao campo de estado, para que o procedimento chamado possa encontrá-las. Por exemplo, considerem a função de biblioteca padrão [illegible] da linguagem C. O primeiro argumento de [illegible] especifica a natureza dos argumentos re-

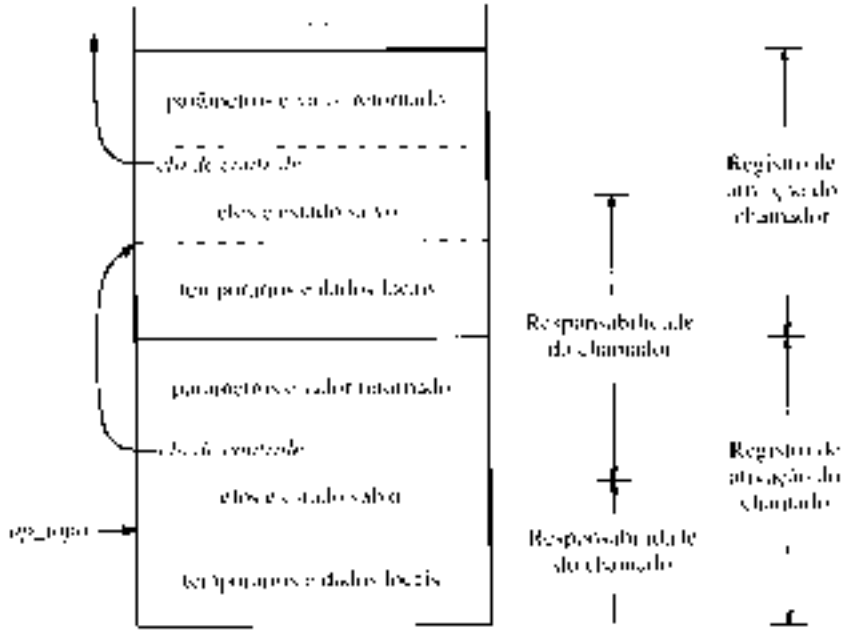


Fig. 7.14. Divisão de tarefas entre chamador e chamado

remanescentes e uma vez que ... tenha encontrado o primeiro argumento, pode localizar os restantes.

Dados de Tamanho Variável

Uma estratégia comum para tratar dados de tamanho variável é sugerida na Fig. 7.15, onde um procedimento p possui três arrays locais. O armazenamento para esses arrays não faz parte do registro de ativação para p; somente um apontador para o início de cada array figura no registro de ativação. Os endereços relativos desses apontadores são conhecidos em tempo de compilação e, por conseguinte, o código-alvo pode ter acesso aos elementos do array através dos apontadores.

Igualmente mostrado na Fig. 7.15 é o procedimento q chamado por p. O registro de ativação para q começa após os arrays de p e os arrays de tamanho variável de q começam além daí.

O acesso aos dados na pilha se dá através de dois apontadores, *topo* e *topo_sp*. O primeiro deles marca o topo atual da pilha; aponta para a posição na qual o próximo registro de ativação irá começar. O segundo é usado para se encontrar os dados locais. Por uma questão de consistência com a organização da Fig. 7.14, suponhamos que *topo_sp* aponte para o final do campo de estado da máquina. Na Fig. 7.15, *topo_sp* aponta para o final desse campo no registro de ativação para q. Dentro desse campo há um elo de controle apontando para o valor prévio de *topo_sp* quando o controle estava na ativação chamadora de q.

O código para reposicionar *topo* e *topo_sp* pode ser gerado em tempo de compilação, usando os tamanhos dos campos nos registros de ativação. Quando q retorna, o novo valor de *topo* é *topo_sp* menos o comprimento dos campos de estado da máquina e de parâmetros no registro de ativação de q. Esse comprimento é conhecido em tempo de compilação, pelo menos pelo chamador. Após ajustar *topo*, o novo valor de *topo_sp* pode ser copiado a partir do elo de controle de q.

Referências Ocas

Sempre que a memória puder ser liberada, emerge o problema das referências ocas. Uma *referência oca* ocorre quando existe uma referência a uma memória que foi liberada. O uso das referências ocas é um erro

Fig. 7.16. Um programa C que deixa p apontando para a memória liberada.

lógico, uma vez que o valor da memória liberada é indefinido de acordo com a semântica da maioria das linguagens de programação. Pior ainda, já que essa memória pode ser posteriormente reservada para outro dado, os programas com referências ocas podem apresentar erros misteriosos.

Exemplo 7.6. O procedimento *dangle* no programa C da Fig. 7.16 retorna um apontador para a memória atribuída ao nome local i. O apontador é criado pelo operador & aplicado a i. Quando o controle retorna para *main* a partir de *dangle*, a memória para os dados locais é liberada e pode ser usada para outros propósitos. Uma vez que p em *main* se refere a essa memória, o uso de p é uma referência oca. □

Um exemplo na Seção 7.7 envolve a liberação sob o controle do programa.

Alocação de Memória *Heap*

A estratégia de alocação de memória de pilha discutida acima não pode ser usada se qualquer das situações for possível:

1. Os valores dos nomes locais precisam ser retidos quando a ativação terminar.
2. A ativação chamada sobrevive ao chamador. Essa possibilidade não pode ocorrer para aquelas linguagens onde as árvores de ativação delineiam corretamente o fluxo de controle entre os procedimentos.

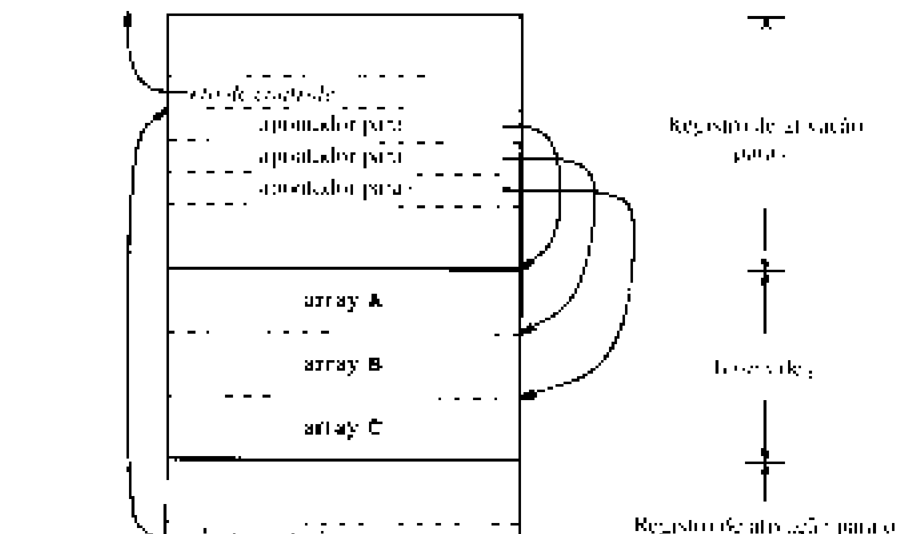


Fig. 7.15. Acesso a arrays alocados dinamicamente.

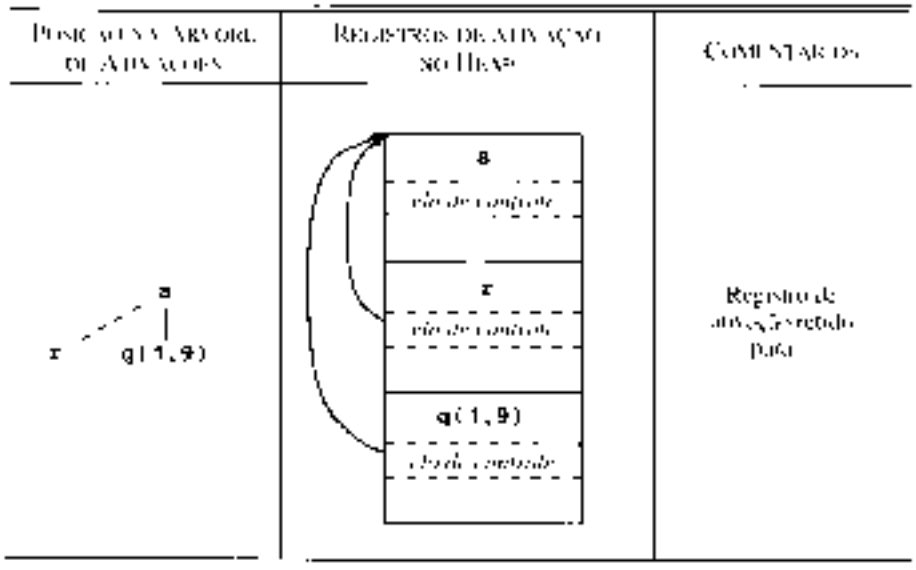


Fig. 7.17. Os registros para as ativações executadas em um ambiente de heap.

Para cada uma das seções anteriores, a liberação dos registros de ativação foi necessária em virtude da validade FIF. Ou seja, que a memória não pode ser organizada como uma pilha.

A alocação de memória em heap fornece blocos de memória de tamanho arbitrário, de acordo com a necessidade, para os registros de ativação em tempo de execução. Assim, as áreas de memória podem ser liberadas em qualquer ordem, de forma que, ao longo do tempo, o heap pode se constituir de áreas livres e ocupadas, intercaladas.

A liberação de uma área de memória ocupada pelo registro de ativação pode ser vista a partir das Figs. 7.17 e 7.13. Na Fig. 7.17, o registro para uma ativação do procedimento *q* retorna ao ponto de chamada. Por conseguinte, o registro para essa ativação de *q* não pode seguir ele mesmo ativo (pode ser liberado, como indicado na Fig. 7.13). Após, o registro de ativação criado para *q* foi liberado, existirá espaço livre adjacente aos registros de ativação para *a* e *x*. Cabe ao gerenciador do heap fazer uso desse espaço.

A questão do gerenciamento eficiente de memória heap é um tópico muito especializado na teoria das estruturas de dados. Algumas técnicas são vistas na Seção 7.8. Neste ponto, destacamos algumas soluções gerais e espaço associados ao uso de um gerenciador de memória heap. Por razões de eficiência, pode ser conveniente manipular registros de ativação ou registros de tamanhos previsíveis como casos especiais, tal como segue:

1. Para cada tamanho específico de interesse, manter uma lista ligada de blocos livres daquele tamanho.
2. Se possível, atender a uma requisição para um tamanho *s* com um bloco de tamanho *s'*, onde *s'* é o menor tamanho maior ou igual a *s*. Quando o bloco for eventualmente liberado, ele retornará à lista ligada da qual é oriundo.
3. Para grandes blocos de memória, usar o gerenciador de heap.

Esse esquema resulta na rápida alocação e liberação de pequenas quantidades de memória, uma vez que a obtenção e os retornos de um bloco a uma de uma lista ligada são operações eficientes. Para grandes quantidades de memória, espera-se que a computação leve algum tempo para explorar a memória e, consequentemente, o tempo gasto pelo alocador é proporcionalmente desprezível comparado ao tempo da computação.

7.4 ACESSO AOS NOMES NÃO LOCAIS

As estratégias de alocação de memória da última seção são adaptadas nesta seção de forma a permitir o acesso aos nomes não locais. Apesar da discussão estar baseada na reserva de memória em pilha para os registros de ativação, as mesmas idéias se aplicam à alocação de memória em heap.

As regras de escopo de uma linguagem determinam a entidade associada com cada nome não local em um programa. Uma regra comum, chamada de regra de escopo léxico ou estático, determina a declaração que se aplica a um nome pela examinação do texto do programa. Pascal, C e Ada são exemplos de muitas linguagens que usam escopo léxico, com a extensão adicional do mecanismo de ocultamento explícito, como será visto abaixo. Uma regra alternativa, chamada de regra de escopo dinâmico, determina a declaração aplicável a um nome em tempo de execução pela consideração das ativações correntes. Lisp, Apl e Snobol estão entre as linguagens que usam o escopo dinâmico.

Começamos pelos blocos e a regra de ocultamento. Mais adiante, consideraremos, então, os tipos mais locais em linguagens como C, onde os escopos são léxicos, todos os nomes pós-locais podem ser alocados a nível externo à escala estaticamente e qualquer declaração associada de procedimento é permitida.

Em linguagens como Pascal, que possuem procedimentos aninhados e escopo léxico, os nomes pós-locais pertencentes a possíveis procedimentos diferentes podem ser parte do ambiente em um dado instante. Discutimos duas formas de se encontrar os registros de ativação contendo memória alocada aos nomes não locais: elos de acesso e exibições.

Uma solução final discute a implantação de escopo dinâmico.

Blocos

Um bloco é uma sentença contendo suas próprias declarações de dados locais. O conceito de bloco foi originado em Algol. Em C, um bloco possui a sintaxe:

{ *declarações comandos* }

Uma característica dos blocos é a sua estrutura de aninhamento. Os delimitadores marcam o início e o final de um bloco. C utiliza chaves, { e }, enquanto que a maioria das Algol-like usa begin e end. Os delimitadores asseguram que um bloco ou é independente ou está totalmente contido dentro de um outro bloco, isto é, pois se dois blocos B_1 e B_2 se sobrepõem, de forma que o primeiro, B_1, começa, em seguida B_2, mas que B_1 termine antes de B_2, terminar, isso proporcionaria um aninhamento e algumas vezes referenciada como *estrutura de bloco*.

O escopo de uma declaração para uma linguagem estruturada em blocos é dado pela *regra de aninhamento mais próximo*:

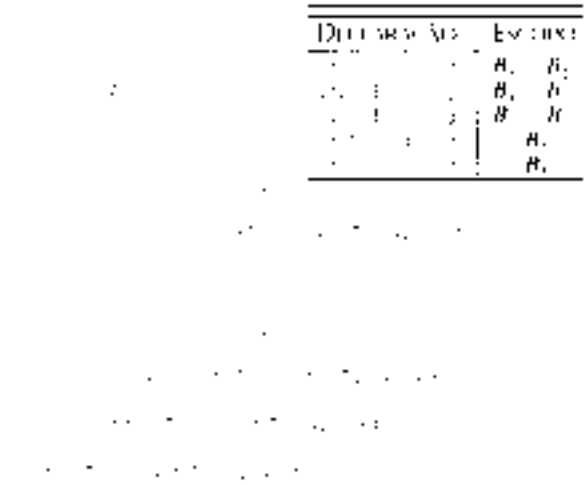


Fig. 7.18. Blocos num programa C

1. O escopo de uma declaração num bloco B inclui B.
2. Se um nome x não for declarado num bloco B, uma ocorrência de x em B está no escopo de uma declaração de x num bloco envolvente B' tal que
   a) B' possui uma declaração para x, e
   b) B' é o bloco mais interiormente aninhado envolvendo B em relação a qualquer outro bloco que contenha uma declaração para x.

Por projeto, cada declaração da Fig. 7.18 inicializa esta nova declaração com o número de blocos ao qual ela mesma aparece. O escopo da declaração de x em B_1 não inclui B_2, porque x é redeclarado em B_2, indicado por $B_1 - B_2$ na figura. Uma tal ausência é chamada de um *buraco* no escopo de uma declaração.

A regra do aninhamento mais interno é refletida pela saída do programa da Fig. 7.18. O comando `printf` é executado a partir de um ponto exatamente antes do mesmo e final a partir dos blocos para o ponto exatamente após o mesmo no texto do fonte. Os argumentos de impressão são, em sequência, encontrados na ordem B_2, B_3, B_1 e B_4, a ordem na qual as variáveis declaradas nos blocos. Os valores de x em cada um dos blocos são:

A extensão de blocos pode ser implementada na mesma maneira alocação de memória de pilha. Uma vez que o escopo de uma declaração não se estende para fora do bloco, qual aparece, o espaço para o nome declarado pode ser reservado quando o controle entra no bloco e liberado quando deixa-o. Essa visão trata os blocos como "procedimentos sem parâmetros", chamados somente a partir do ponto exatamente antes do bloco e retornando somente para o ponto exatamente após o mesmo. O ambiente não local para um bloco pode ser mantido exatamente as mesmas maneiras que os procedimentos aninhados, como discutido na Seção 7.3. Note, entretanto, que os blocos são mais simples do que os procedimentos, porque não há parâmetros a serem passados e nem uso de comando para chamar blocos a partir de blocos segue o texto do programa.[1]

[1] Uma alternativa à extensão de blocos é tratar cada bloco como um procedimento sem parâmetros, que é chamado a partir do ponto imediatamente antes do bloco. Variáveis não locais referem-se ao bloco imediatamente envolvente, como de costume. Os procedimentos são tratados como discutido na Seção 7.3. A semântica da repetição de entrada e saída de blocos é, então, a mesma que a de procedimentos aninhados.

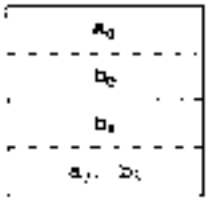


Fig. 7.19. Armazenamento para blocos declarados na Fig. 7.18.

Uma implementação alternativa consiste em se reservar memória para todo o corpo de um procedimento de uma só vez. Se existirem blocos dentro de um procedimento, o espaço para o armazenamento necessário pelas declarações dentro dos blocos, Para o bloco B_2 na Fig. 7.18, podemos reservar memória como na Fig. 7.19. Os subscritos dos objetos de dados locais para x identificam os blocos nos quais são declarados. Note que x_2 e x_3 podem ser associados a uma mesma memória porque estão em blocos que não estão ativos ao mesmo tempo.

Na ausência de dados de tamanho variável, a quantidade máxima de memória necessitada durante a execução de um bloco pode ser determinada em tempo de compilação (dados de tamanho variável podem ser tratados usando-se apontadores como na Seção 7.7). Assegurando essa determinação, assumimos conservativamente que todos os possíveis caminhos de um programa passam de fato ser executados. Isto é, assumimos que todas as partes de uma instrução `if` possam ser executadas e que todos os comandos dentro de laços `while` possam ser executados.

Escopo Léxico sem Procedimentos Aninhados

As regras de escopo estático para C são mais simples do que aquelas para Pascal discutidas em seguida porque as definições de procedimentos não podem ser aninhadas em C. Isto é, uma definição de procedimento não pode ocorrer dentro de outra. Como na Fig. 7.20, um programa C consiste em uma sequência de declarações de variáveis e procedimentos. Se uma chamada de função f existir em algum lugar no escopo de uma declaração global ou externa de um nome x — em alguma função — possivelmente declarada em qualquer função, o escopo de uma declaração externa de x em uma função consiste nos corpos das funções que seguem a declaração, exceto naquelas em que x for redeclarado dentro de uma função. Na Fig. 7.20, as ocorrências não locais de x [illegible] referem-se à declaração global x.

Fig. 7.20. Variáveis não locais em C.

Na ausência de procedimentos aninhados, a estratégia de alocação de pilha da Fig. 7.3, para os nomes locais, pode ser usada diretamente para uma linguagem estruturada em escopo léxico como C. O armazenamento para todos os nomes declarados fora de quaisquer procedimentos pode ser alocado estaticamente. A posição desse armazenamento é conhecida em tempo de compilação e, dessa forma, se um nome for não local no corpo de um procedimento, usamos simplesmente o endereço determinado estaticamente. Qualquer outro nome não local deve estar na pilha, acessível através do ponteiro *top*. Os procedimentos aninhados fazem o escopo não local para o nome não local pode estar relacionado a dados na pilha como será discutido abaixo.

Um benefício importante da alocação estática para nomes não locais está em que os procedimentos podem ser transmitidos como

...

Fig. 7.21. Programa Pascal com escopo estático léxico.

...

Escopo Léxico com Procedimentos Aninhados

...

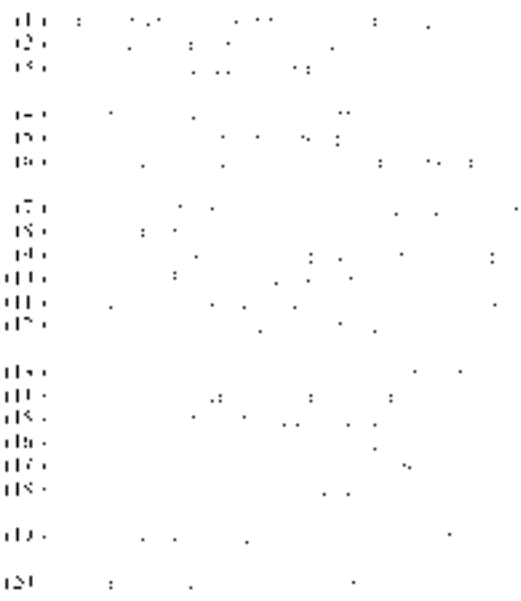


Fig. 7.22. Programa Pascal com procedimentos aninhados.

...

Profundidade de Aninhamento

...

Endereços

...

1. Quando ocorrer a ativação de um procedimento, um registro de ativação para ele será empilhado. Seguindo o elo de acesso desse registro, pode-se encontrar o registro de ativação mais recente do procedimento que o declarou.

acesso existe, então um elo pode ser seguido realizando uma operação simples de indexação.

2. Após seguir n elos de acesso, apontamos um registro de ativação para o procedimento no qual x é local. Como discutido na última seção, seu armazenamento estará a um deslocamento fixo relativo a posição do registro. Em particular, o deslocamento pode ser relativo ao elo de acesso.

Portanto, quando o endereço associado a um nome não local x num procedimento p é calculado pelo seguinte par pré-computado em tempo de compilação e armazenado na tabela de símbolos

(n, s) = (deslocamento dentro do registro de ativação contendo x

O primeiro componente n é o número de elos de acesso a serem atravessados.

Por exemplo, as linhas 15-16 da Fig. 7.22, o procedimento p está à profundidade de aninhamento 3. As referências a nomes não locais m e n são às profundidades de aninhamento 1 e 2, respectivamente. Os registros de ativação contendo m e n são, portanto, encontrados seguindo-se $3-1=2$ e $3-2=1$ elos de acesso, respectivamente, a partir do registro de ativação de p.

O código para estabelecer os elos de acesso é parte da sequência de chamada. Suponha, novamente, que o procedimento p, com profundidade de aninhamento n_p, chame o procedimento q, à profundidade de aninhamento n_q. O código para estabelecer o elo de acesso no procedimento chamado depende de se o nível n_q está aninhado ou não no chamador p.

1. Caso $n_q > n_p$. Isto só ocorre quando o procedimento chamado q está aninhado mais profundamente do que p, de forma que p deve estar dentro do(s) procedimento(s) que estão acessíveis a p. Este caso ocorre quando p chama q em p na Fig. 7.23(a) e quando p chama q em r na Fig. 7.23(c). Nesse caso, o elo de acesso no procedimento chamado q terá de apontar para o elo de acesso no registro de ativação do chamador, exatamente abaixo na pilha.
2. Caso $n_q = n_p$. A partir das regras de escopo, os procedimentos envolvendo as profundidades de aninhamento 1, 2, $\ldots$, $n_p - 1$, a partir dos procedimentos chamado e chamador, precisam ser os mesmos, exceto quando p chama a si mesmo, como na Fig. 7.23(a) e quando p chama q em q na Fig. 7.23(d). Seguindo-se $n_p - n_q + 1$ elos de acesso a partir do chamador, atingimos o registro de ativação mais recente do procedimento que envolve estaticamente mais proximamente tanto o procedimento chamado quanto o chamador. O elo de acesso atingido é aquele para o qual o elo de acesso no procedimento chamado precisará apontar. De novo, $n_p - n_q + 1$ pode ser computado em tempo de compilação.

Parâmetros Tipo Procedimento

As regras de escopo léxico se aplicam mesmo quando um procedimento aninhado é passado como parâmetro. A função p nas linhas 6-7 do programa Pascal na Fig. 7.24, por exemplo, é um nome não local dentro do acesso às variáveis de x estão circuladas na figura. A linha 8, o procedimento anônimo $f(a)$ é então passado como parâmetro a c. Note que, nesta linha da declaração de c, a linha 5 não está no escopo de x nas linhas 2-3

Fig. 7.24. Um elo de acesso precisa ser passado com o parâmetro $f(a)$.

Dentro do corpo de p, o enunciado $c(0)$ na linha 9 chama $f(a)$ porque o parâmetro formal c se refere a $f(a)$. Imprime o resultado da chamada $f(a)$.

Como estabelecemos o elo de acesso para a ativação de $f(a)$ resposta é que um procedimento aninhado, que seja passado como parâmetro, precisa carregar seu elo de acesso junto com a instância na Fig. 7.25. Quando o procedimento f passa f determinamos o elo de acesso para f como faríamos se estivesse chamando f. Esse elo é passado junto com f para c. Subsequentemente, quando f for ativado, o ponto de f é usado para estabelecer o elo de acesso no registro de ativação para f.

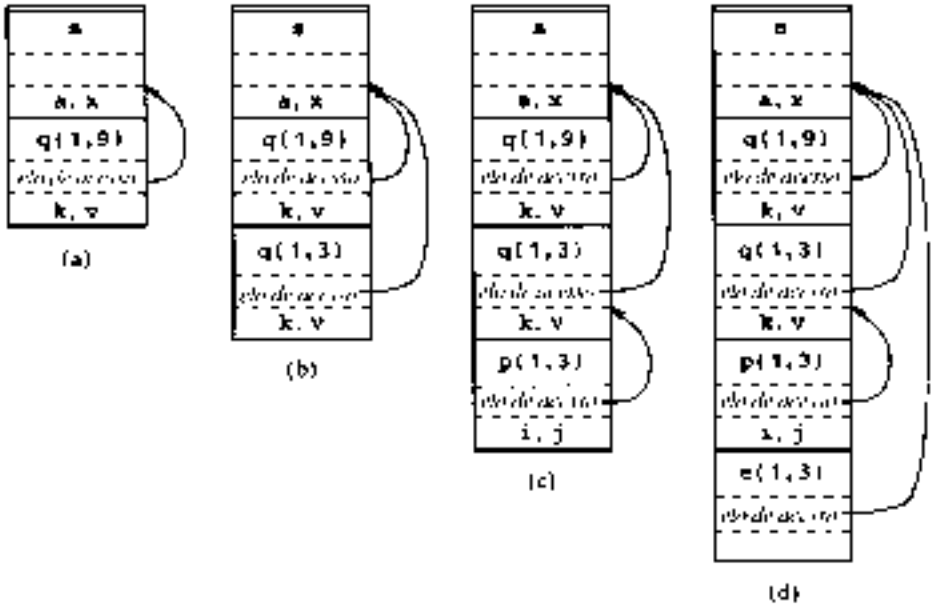


Fig. 7.23. Elos de acesso para escopo com blocos aninhados.

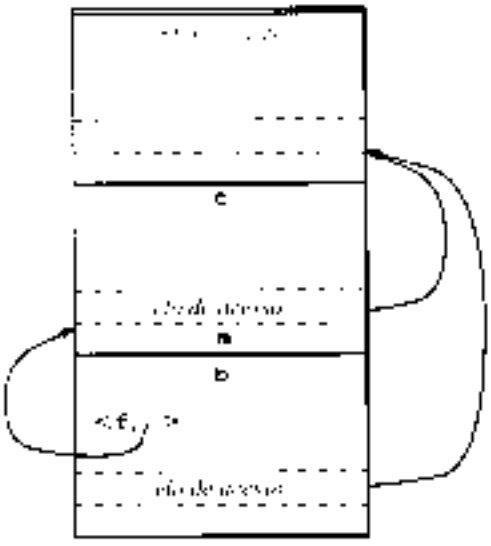


Fig. 7.25 Cadeia encadeada que provê um caminho para cada elo de acesso

Avaliação

Intuitivamente, o método descrito acima sugere que o cálculo através de um elo de acesso pode ser simplificado usando-se um *array* de apontadores para os registros de ativação, chamado *display*. Mantemos o *array* de tal forma que a memória para uma dada local à profundidade de aninhamento i esteja no registro de ativação apontado pelo elemento $d[i]$ do *display*.

Suponhamos que o controle esteja na ativação de um procedimento p a uma profundidade de aninhamento i. Então os primeiros $i-1$ elementos do *display* apontam para as ativações dos procedimentos que envolvem lexicamente o procedimento p, e $d[i]$ aponta para a ativação de p. Usar um *display* é geralmente mais rápido do que seguir os elos de acesso, porque o registro de ativação que abriga uma não local é encontrado através de um acesso a um elemento de d, ou através de se seguir exatamente um apontador.

Um método simples para se manter o *display* usa elos de acesso adicionalmente ao *display*. Como parte das seqüências de chamada e retorno, o *display* é atualizado de acordo com a cadeia de elos de acesso. Quando um elo para um registro de ativação à profundidade i é estabelecido, o elemento de *display* $d[i]$ é estabelecido de forma a apontar para o registro de ativação. Com efeito, o *display* duplica as informações na cadeia de elos de acesso.

Um arranjo simples acima pode ser melhorado. O método ilustrado na Fig. 7.26 requer menos trabalho a entrada e à saída de um procedimento, porque o usual em que os procedimentos locais são passados como parâmetros. Na Fig. 7.26, o *display* consiste em um *array* global, separado da pilha. Os quantificadores na figura se referem à execução do texto-fonte na Fig. 7.22. De novo, somente a primeira letra de cada procedimento é mostrada.

A Fig. 7.26(a) mostra a situação imediatamente antes da ativação [illegible] começar. Como [illegible] está à profundidade de aninhamento 2, o elemento de [illegible] $d[2]$ é afetado quando uma ativação de [illegible] começa. O efeito da ativação [illegible] sobre $d[2]$ é mostrado na Fig. 7.26(b), onde $d[2]$ agora aponta para um

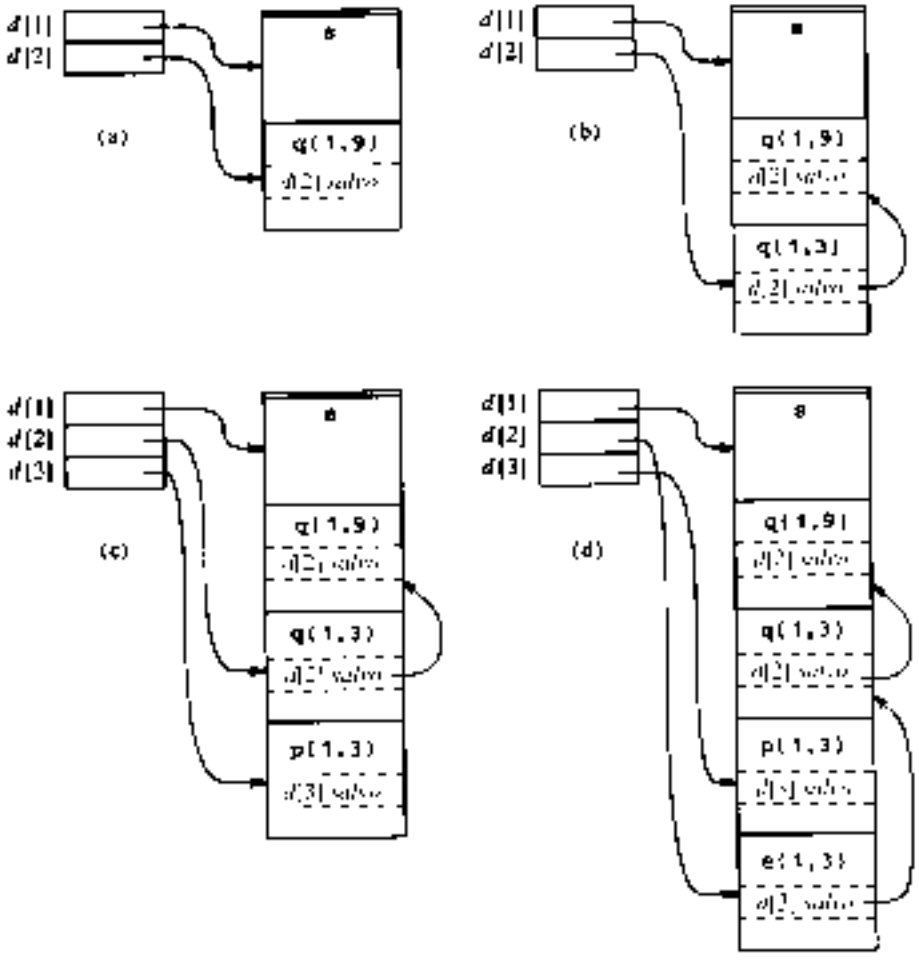


Fig. 7.26 Mantendo o *display* quando procedimentos não são transmitidos como parâmetros

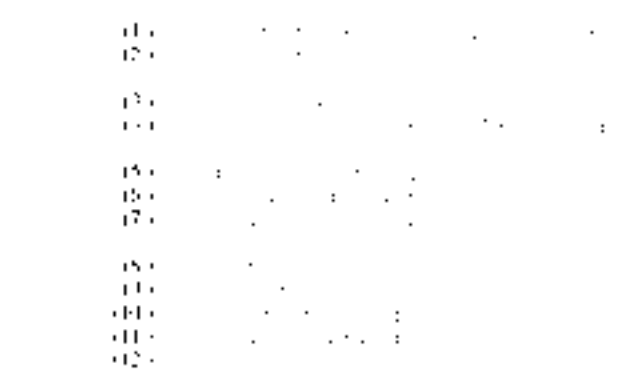


Fig. 7.27. Variável dependente em tempo de execução, escopo léxico e dinâmico

Escopo Dinâmico

Sob o escopo dinâmico, uma nova ativação herda as ligações existentes para nomes não locais no momento. Um nome não local em uma ativação chamada se refere à mesma instância que tinha associada quando na ativação chamadora. As novas associações são estabelecidas para os nomes locais dos procedimentos chamados; os nomes se referem a membros do novo registro de ativação.

O programa na Fig. 7.27 ilustra o escopo dinâmico. O procedimento nas linhas 3–4 usa variáveis não locais. Sob escopo léxico em Pascal, o não local é sempre a variável declarada na linha 2, de forma que a saída do programa é

```
1
2
```

Entretanto, sob o escopo dinâmico, a saída é

```
1
2
```

Quando é chamado nas linhas 10–11 do programa principal, o escopo pega a variável local do programa principal declarada. Entretanto, quando é chamado na linha 7, ele usa a variável local de , escrita porque a variável local de é usada.

Os dois enfoques seguintes para implementar o escopo dinâmico têm alguma semelhança com os usos dos elos de acesso e *display*, respectivamente, na implementação do escopo léxico.

1. *Acesso profundo.* Conceitualmente, o escopo dinâmico resulta se os elos de acesso apontarem para os ponteiros registros de ativação que os elos de controle apontam. Uma implementação simples é se dispensar os elos de acesso e se utilizar os elos de controle para pesquisar na pilha, usando pelo primeiro registro de ativação que contenha a memória para o nome local. Diferentes ocorrências podem ocorrer do fato da procura poder ser "mais funda" na pilha. A profundidade de uma ativação pode ir dependendo da entrada para o programa e não pode ser determinada em tempo de compilação.
2. *Acesso superficial.* Aqui a idéia é guardar o valor corrente de cada nome em memória estaticamente reservada. Quando ocorre uma nova ativação de um procedimento com um nome local, ele ocupa memória nesse local estaticamente pré-alocada. O valor prévio desse nome é salvo no registro de ativação para o procedimento ser restaurado quando a ativação do termina.

Novamente, o valor de [i] no novo registro de ativação
Fazemos [i] apontar para o novo registro de ativação

Evidentemente, antes da ativação terminar, [i] é restabelecido com o valor salvo.

Esses passos são justificados como se segue. Suponhamos que um procedimento *p* tenha localidade de aninhamento *y* e chame um procedimento *q* de localidade *x*. Devem ocorrer dois casos, dependendo do procedimento chamado estar ou não aninhado no chamador (ou seja, no escopo do programa, como na discussão dos elos de acesso).

1. Caso $y \geq x$ (chamador = p, y; chamado = q, x). Escopo procedimentos aninhados estão aninhados dentro de localização. Os nomes no elemento do *display* correspondente neste não precisam ser modificados e fazemos [i] apontar para a nova região de ativação. Esse caso é ilustrado na Fig. 7.26(a), quando é chamado e também quando é chamado (ver Fig. 7.26(c)).
2. Caso $y < x$. De novo, os procedimentos envolventes são profundidades $1, 2, \ldots, y$. Não precisamos ajustar o chamado e os chamadores precisam ser os mesmos. Aqui precisamos guardar o valor atual de [i] no novo registro de ativação e fazer [i] apontar para esse mesmo novo registro. O *display* é atualizado corretamente porque os procedimentos $1, 2, \ldots, y$ são os mesmos como antes.

Um exemplo do Caso 2 com $y = 1$ ocorre quando é chamado na Fig. 7.26(b). Um exemplo mais interessante ocorre quando a ativação de , a profundidade de aninhamento 3, chama , a profundidade 2 e o procedimento envolvente de ambos é , à profundidade 1, como na Fig. 7.26(d) do programa esboçado na Fig. 7.25. Note-se que quando é chamado [3] pertence a , chamada através de , mas ainda está acessível, apesar de não poder ser acessado mesmo quando o controle estiver em . Chamar é um outro procedimento à profundidade 3, e aquele procedimento armazena [3] e o restaura antes de retornar para . Podemos, por conseguinte, mostrar que cada procedimento vê o *display* correto para todas as profundidades de sua própria.

Existem casos legais onde um *display* pode ser mantido. Se existem registradores suficientes, o *display* descritivo mencionado pode ser uma coleção de registradores. Note-se que o compilador pode determinar o tamanho máximo desse vetor a partir de localização para de aninhamento entre dois procedimentos no programa. De outra forma, o *display* pode ser mantido em memória estaticamente alocada e todas as referências aos registros de ativação começam utilizando um endereçamento indireto através do apontador apropriado do *display*. Esse esquema é razoável numa máquina com endereçamento indireto, apesar de cada indireção custar um ciclo de memória. Outra possibilidade é armazenar o *display* na pilha em tempo de execução e criar uma nova cópia do mesmo a cada entrada de procedimento.

Escopo Dinâmico

Sob o escopo dinâmico, uma nova ativação herda as ligações existentes para nomes não locais no momento. Um nome não local em uma ativação chamada se refere à mesma instância que tinha associada quando na ativação chamadora. As novas associações são estabelecidas para os nomes locais dos procedimentos chamados; os nomes se referem a membros do novo registro de ativação.

O programa na Fig. 7.27 ilustra o escopo dinâmico. O procedimento nas linhas 3–4 usa variáveis não locais. Sob escopo léxico em Pascal, o não local é sempre a variável declarada na linha 2, de forma que a saída do programa é

```
1
2
```

Entretanto, sob o escopo dinâmico, a saída é

```
1
2
```

Quando é chamado nas linhas 10–11 do programa principal, o escopo pega a variável local do programa principal declarada. Entretanto, quando é chamado na linha 7, ele usa a variável local de , escrita porque a variável local de é usada.

Os dois enfoques seguintes para implementar o escopo dinâmico têm alguma semelhança com os usos dos elos de acesso e *display*, respectivamente, na implementação do escopo léxico.

1. *Acesso profundo.* Conceitualmente, o escopo dinâmico resulta se os elos de acesso apontarem para os ponteiros registros de ativação que os elos de controle apontam. Uma implementação simples é se dispensar os elos de acesso e se utilizar os elos de controle para pesquisar na pilha, usando pelo primeiro registro de ativação que contenha a memória para o nome local. Diferentes ocorrências podem ocorrer do fato da procura poder ser "mais funda" na pilha. A profundidade de uma ativação pode ir dependendo da entrada para o programa e não pode ser determinada em tempo de compilação.
2. *Acesso superficial.* Aqui a idéia é guardar o valor corrente de cada nome em memória estaticamente reservada. Quando ocorre uma nova ativação de um procedimento com um nome local, ele ocupa memória nesse local estaticamente pré-alocada. O valor prévio desse nome é salvo no registro de ativação para o procedimento ser restaurado quando a ativação do termina.

...depende de quão distantes estão um do outro o acesso profundo desejado e mais tempo para se ter acesso a uma remissão local mais rápida. Isso ...

7.5 TRANSMISSÃO DE PARÂMETROS

Quando um procedimento chama outro, o método usual de comunicação ... entre os parâmetros e argumentos de nomes não locais ... procedimento chamado. Ambos fornecem meios de passar parâmetros. São ... pelo procedimento da Fig. 7.28 para imprimir os valores de ... de ... Aqui, o procedimento não local ... [illegible]

```
(1) ...
(2) ...
(3) ...
(4) ...
(5) ...
```

Fig. 7.28 Um procedimento Pascal ... com nove métodos de parâmetros.

Vários métodos comuns para associar os parâmetros atuais e formais são discutidos nesta seção. São chamados por valor, referência, cópia e restauração, nome e por especificação formal. Um programa ... especifica o método de transmissão de parâmetros que uma linguagem ou compilador usa, porque o resultado de um programa pode depender do método usado.

Por que tantos métodos? Os diferentes métodos emergem da ... diferentes interpretações do que uma expressão representa. Num comando como

```
...
```

a expressão ... representa um valor enquanto que a ... representa um local ... na qual o valor de ... é colocado. A decisão de se usar a localização ou o valor representado por uma expressão é determinada dependendo de ... e ... [illegible] especificado no Capítulo 2, o termo variável se refere à memória representada por uma expressão e valor ao valor contido na região de memória. Os termos *l-value* e *r-value* foram cunhados respectivamente para designar objetos à direita e à esquerda de uma atribuição.

As diferenças entre os métodos de passagem de parâmetros estão ... Essas diferenças ocorrem porque o parâmetro atual representa ... um valor e uma variável ... o texto do próprio parâmetro atual.

Chamada por Valor

Este é, com certo sentido, o método mais simples possível de se transmitir parâmetros. Os parâmetros atuais são avaliados e seus valores são passados para o procedimento chamado. A chamada por valor é usada em C e os parâmetros em Pascal são explicitamente restringidos dessa forma. Todos os programas deste capítulo representam sobre esse método para a transmissão de parâmetros. A transmissão por valor pode ser implementada da forma seguinte:

1. Um parâmetro formal é tratado exatamente como um nome local ... de forma que a memória para os parâmetros formais está no registro de ativação do procedimento chamado.
2. O chamador avalia os parâmetros atuais e coloca os valores ... na memória para os parâmetros formais.

Uma ligeira distinção da transmissão por valor está em que os parâmetros formais ... não alteram os valores no registro de ativação do chamador. Se a palavra chave ... à linha 3 da Fig. 7.29 for omitida, Pascal irá transportar ... por valor para o procedimento ... A chamada ... à linha 12 deixará os valores de ... inalterados. Sob a transmissão por valor, o efeito da chamada ... é equivalente à sequência de passos

```
...
...
...
...
...
```

onde ... são locais ... Apesar dessas atribuições mudarem os valores dos locais ... as mudanças estarão perdidas quando o controle retornar da chamada, o registro de ativação de ... for liberado. As chamadas, portanto, não podem alterar os registros de ativação do chamador.

```
(1) ...
(2) ...
(3) ...
(4) ...
(5) ...
(6) ...
(7) ...
(8) ...
(9) ...

(10) ...
(11) ...
(12) ...
(13) ...
(14) ...
```

Fig. 7.29 Um programa Pascal com o procedimento ...

Um procedimento chamado por valor somente pode alterar seu chamador através de nomes não locais ... na Fig. 7.35 ... variáveis apontadoras que sejam explicitamente passadas como valores. No programa C da Fig. 7.30, ... são declarados à linha 2 como apontadores para inteiros; o operador ... na chamada ... à linha 8 resulta nos apontadores para ... sendo passados para ... A saída desse programa é

```
...
```

Uma dos apontadores neste exemplo sugere como o compilador usando uma chamada por referência irá buscar os valores

```
(1) ...
(2) ...
(3) ...
(4) ...
(5) ...
(6) ...
(7) ...
(8) ...
(9) ...
(10) ...
```

Fig. 7.30 Programa C usando apontadores num procedimento chamado por valor.

Chamada por Referência

Quando o parâmetro formal é sinalizado por *referência*, também conhecida como *variável* ou *endereço*, ou *chamada por localização*, o chamador passa para o procedimento chamado um apontador para o endereço de armazenamento de cada parâmetro atual.

1. Se o parâmetro atual for um nome ou uma expressão contendo um valor-l, então aquele valor-l é passado.
2. Se, por outro lado, o parâmetro atual for uma expressão como `a + b` que não tem valor-l, a expressão é avaliada normalmente, a localização é reservada, e depois a localização é passada.

Uma referência a um parâmetro formal no procedimento chamado se torna um endereço indireto através do apontador para o parâmetro correspondente no procedimento chamado.

Exemplo 7.7. Considere o procedimento `swap` da Fig. 7.2b. Uma chamada para `swap` com parâmetros atuais `i` e `a[i]` faz com que o procedimento `swap` execute efetivamente as seguintes sequências de passos:

1. Copiar o endereço base de `a` e o valor de `i` no registro de ativação do procedimento chamado. Chamemos essas localizações `x` e `y`, respectivamente.
2. Estabelecer `t` como o valor contido na localização apontada por `x` (isto é, `t = a[i]`). Esse passo corresponde à primeira linha de `swap`.
3. Estabelecer o conteúdo da localização apontada por `x` como o valor da localização apontada por `y` (isto é, `a[i] = a[i]`). Esse passo corresponde à segunda linha de `swap`.
4. Estabelecer o conteúdo da localização apontada por `y` como um valor igual a `t` (isto é, fazer `a[i] = t`). Esse passo corresponde à terceira linha de `swap`.

A chamada por referência é usada numa série de linguagens nos parâmetros [illegible] em Pascal são passados dessa forma. Os arrays são tradicionalmente passados por referência.

Cópia-e-Restauração

Um sistema híbrido entre a chamada por valor e a por referência é a *cópia-e-restauração*, também conhecida como *cópia para dentro e cópia para fora* ou *valor-resultado*.

1. Antes de o controle fluir para o procedimento chamado, os parâmetros atuais são avaliados. Os valores-r dos parâmetros atuais são passados para o procedimento chamado como na transmissão por valor. Adicionalmente, os valores-l daqueles parâmetros atuais que tenham valores-l são determinados antes da chamada.
2. Quando o controle retorna, os valores correntes dos parâmetros formais são copiados de volta dentro dos valores-l dos parâmetros atuais, usando os valores-l computados antes da chamada. Somente os parâmetros atuais que tenham valores-l são copiados de volta dessa maneira.

O primeiro passo, "cópia para dentro", na ativação do procedimento chamado, atribui os valores dos parâmetros atuais correspondentes para os parâmetros formais. O segundo passo, "cópia para fora", no registro de ativação do procedimento chamador, usa os valores finais dos parâmetros formais idênticos aos valores-l computados a partir dos parâmetros atuais antes da chamada.

Note que [illegible] trabalha corretamente usando a cópia-e-restauração, uma vez que a localização de `a[i]` é passada e preservada pelo programa chamador antes da chamada. Por consequência, o valor final do parâmetro formal `y` que será o valor inicial de `a[i]` é computado na localização correta, ainda que o valor de `i` seja modificado pelo chamado (porque o valor de `i` é usado).

Fig. 7.31. [illegible]

A cópia-e-restauração é usada em algumas implementações Fortran. No entanto, outras implementações usam a chamada por referência. Avaliadores que entrelaçam para implementar o procedimento chamado possuem mais de uma forma de acessar um local do registro de ativação do procedimento chamador. A variável `a` estabelecida pela chamada [illegible] na Fig. 7.31 pode ter acesso a ela como um nome local através do parâmetro formal `x`. Sob a chamada por referência, as atribuições [illegible] afetam imediatamente `a[i]`, de forma que o valor final de `i` é usado. Sob cópia-e-restauração, contudo, o valor do parâmetro atual `i` é copiado no parâmetro formal `y`. O valor final de `y` é copiado de volta dentro do valor-l exatamente antes do controle retornar, de forma que o valor final de `i` é [illegible].

Chamada por Nome

A chamada por nome é definida tradicionalmente pela *regra de cópia* de Algol, a qual é:

1. O procedimento é tratado como se fosse uma macro; isto é, ele é substituído pela chamada feita no chamador, com os parâmetros atuais substituídos literalmente pelos parâmetros formais. Tal substituição literal é chamada de *expansão de macro* ou *expansão in-line*.
2. Os nomes locais no procedimento chamado são mantidos distintos dos nomes do procedimento chamador. Podemos pensar em cada nome local no procedimento chamado como sistematicamente renomeado para um nome distinto antes da expansão da macro ser feita.
3. Os parâmetros atuais são envolvidos por parênteses, se necessário, para preservar as suas integridades.

Exemplo 7.8. A chamada [illegible] proveniente do Exemplo 7.7 seria implementada como se fosse

[illegible]

Por conseguinte, se `a[i]` é chamada por nome, `i` é estabelecido como valor de `x`, mas pessoas não esperam o resultado desse estabelecimento [illegible] — as linhas de `a[i] = t` em `swap`, onde `t` é o valor inicial de `i`.

Esse fenômeno ocorre porque a localização de uma atribuição [illegible] de `i` não é a mesma após a chamada que seja necessária, porque o valor de `i` já mudou. Uma versão operacional de [illegible] geralmente pode ser escrita se a chamada por nome for usada [illegible] (Ingerman 1970).

Apesar de a chamada por nome ser primariamente de interesse teórico, a técnica conceitualmente relacionada da expansão em tempo de execução foi sugerida para reduzir o tempo de execução de um programa. Existe um certo custo associado ao estabelecimento de um procedimento — espaço é reservado para o registro de ativação, as variáveis locais são alocadas e os valores estabelecidos e, em segunda, o controle é

[illegible]

Exemplo 7.9. [illegible]

[illegible]

7.6 TABELAS DE SÍMBOLOS

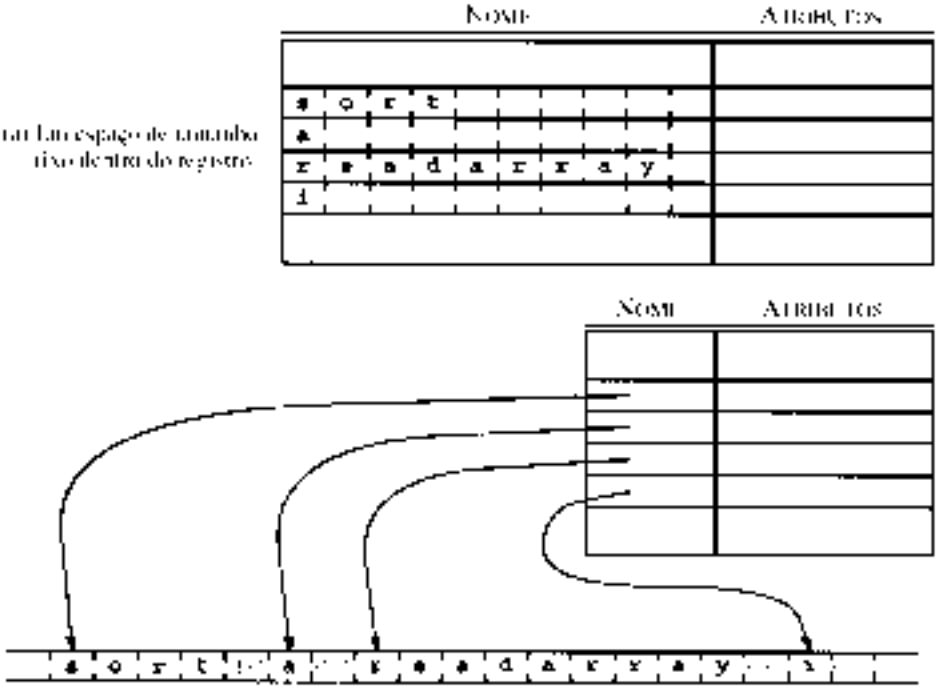
[illegible]

Entradas da Tabela de Símbolos

[illegible]

Caracteres num Nome

[illegible]



Nota: EOC simboliza um caractere de fim de cadeia. (b) Nomes variáveis apontados

Fig. 7.32. Armazenamento de nomes de comprimento

um nome e um apontador para a entrada da tabela de símbolos para o mesmo.

Se existir um limite superior moderado no comprimento dos nomes, os caracteres de cada nome podem ser armazenados na entrada da tabela de símbolos, como na Fig. 7.32(a). Se não existir limite no tamanho de um nome, como é mais frequentemente o caso, o esquema indicado na Fig. 7.32(b) pode ser usado. Assim que se localizar cada entrada da tabela de símbolos, apontando para uma área de espaço para abrigar um lexema, podemos utilizar o espaço mais eficientemente se existir espaço somente para o apontador nos campos da tabela de símbolos. Num registro para um nome, colocamos um apontador para um array de caracteres seguido de um *caractere de fim de cadeia*, marcando a posição do primeiro caractere do lexema. O esquema indicado na Fig. 7.32(b) permite que o tamanho do campo nome, da entrada da tabela de símbolos, permaneça uma constante.

Um esquema completo para armazenar os nomes pode ser armazenado, a fim de assegurar que todos os usos do mesmo nome quando ocorrerem estejam associados ao mesmo registro da tabela de símbolos. Precisamos, entretanto, distinguir entre as ocorrências de um mesmo lexema que estejam em escopos de declarações diferentes.

Informações de Alocação de Memória

As informações a respeito da localização de memória que serão associadas aos nomes em tempo de execução são incluídas na tabela de símbolos. Os endereços relativos simples podem ser mantidos estáticos. Se o código alvo estiver em linguagem de montagem, podemos fazer com que o montador cuide das localizações de memória para as variáveis nomeadas. Tudo o que temos de fazer é espalhar a tabela de símbolos, após gerar o código de montagem do programa, e adicionar as definições de dados para cada nome no programa em linguagem de montagem.

Se, entretanto, o compilador deve gerar código de máquina, a posição de cada objeto de dados relativa a uma área particular, tal como o início de um registro de ativação, precisa ser avaliada. O mesmo mecanismo se aplica a um bloco de dados carregado como um módulo separado do programa. Por exemplo, os blocos COMMON em Fortran são carregados separadamente e as posições dos nomes relativas ao início do bloco no qual residem precisam ser determinadas. Pelas razões descritas, na Seção 7.9, e também na Seção 6.3 precisamos também tratar de um problema em Fortran, a medida em que precisamos atribuir declarações para os nomes de forma que todas as declarações para um procedimento tenham sido examinadas e os nomes declarados [illegible] tenham sido processados.

No caso de nomes cuja memória seja alocada numa pilha de dados, o compilador não tem acesso a qualquer área de dados, exceto o registro de ativação para o seu procedimento, como na Seção 7.3.

A Estrutura de Lista para a Tabela de Símbolos

A estrutura mais simples e mais fácil de se implementar para a tabela de símbolos é uma lista linear de registros — mostrada na Fig. 7.33. Usamos um array simples, cada um de modo explícito de tipo, para armazenar os nomes e suas informações associadas. Os novos nomes são acrescentados à lista na ordem em que são encontrados. A posição final do array é marcada pelo apontador *proximo_entrada_disponivel*, que aponta para onde a próxima entrada na tabela de símbolos será colocada. A pesquisa de um nome se dá do final do array para o início. Quando o nome é localizado, as informações associadas podem ser encontradas nas posições adjacentes. Se atingimos o início do array sem ter encontrado o nome, uma entrada nova é criada para o nome e a informação é colocada na posição apontada por *proximo_entrada_disponivel*.

Note-se que a combinação de uma entrada para um nome e a pesquisa do nome da tabela de símbolos são operações independentes — podemos desejar realizar uma sem termos que realizar a outra. Numa linguagem estruturada em blocos a ocorrência de um nome estará no corpo de declaração com a qual ele está mais estreitamente relacionado. Podemos implementar essa regra de escopo mantendo a estrutura de lista de dados através da constatação de que a nova entrada para um nome é a que for mais recentemente criada, até que o mesmo for declarado. Uma nova entrada é feita nas palavras imediatamente seguintes ao apontador *proximo_entrada_disponivel*; esse apontador é incrementado pelo tamanho do registro da tabela de símbolos. Uma vez que essas entradas são inseridas em ordem, começando

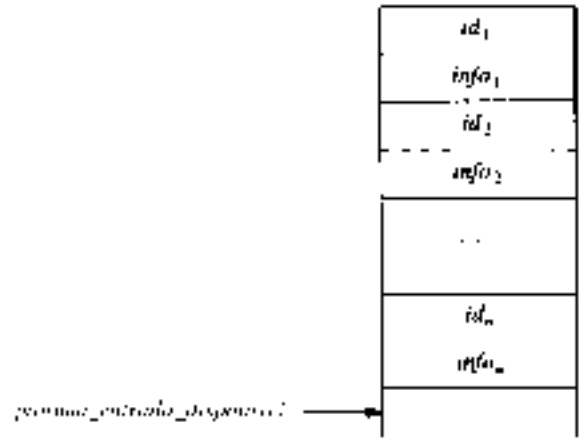


Fig. 7.33. Uma área local de registros

Vetor de cabeçalhos de listas indexados pelo valor de hash

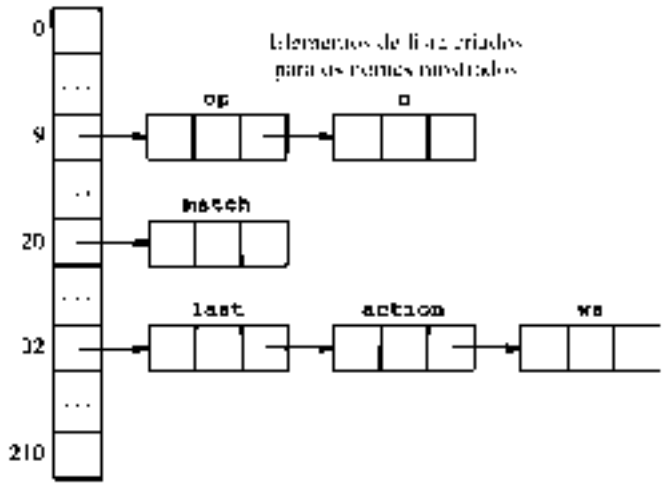


Fig. 7.34. Uma tabela hash de tam. = 211

Cada registro na tabela de símbolos figura exatamente em uma dessas listas. O armazenamento para esses registros pode ser de interesse para o projetista de um gerador de registros como discutido na próxima seção. Alternativamente, as facilidades de alocação dinâmica de memória da linguagem de implementação podem ser usadas para obter espaço para os registros, freqüentemente pelo preço de alguma perda de eficiência.

Para se determinar se existe uma entrada para uma cadeia de caracteres s na tabela de símbolos, aplica-se uma função hash a s de tal forma que ela retorne um inteiro entre 0 e m − 1. Se s existe na tabela de símbolos, estará na lista armazenada por hash[s]. Se s ainda não existe na tabela de símbolos, é inserida através da criação de um registro para ela nessa lista e ligado ao índice da lista armazenado por hash[s].

Quase sempre, a lista média contém um pequeno número de registros, se existirem muitos nomes numa tabela de comprimento m. Pela escolha de uma função hash de tal forma que não exista mais do que uma pequena constante, 2 por exemplo, o tempo para se ter acesso a uma entrada da tabela é essencialmente constante.

O espaço ocupado pela tabela de símbolos consiste em m palavras para a tabela hash e n palavras para as entradas da tabela, onde n é o número de palavras por entrada da tabela. Por conseguinte, o espaço para a tabela hash é dependente somente de m e o espaço para as entradas da tabela depende somente do número de entradas.

A escolha do valor m depende da aplicação pretendida para a tabela de símbolos. Escolhê-lo como sendo de mesma ordem de tamanho que a pesquisa em tabela seja uma função desprezível do tempo total despendido pelo compilador mesmo para programas de tamanho moderado. Quando o método para o compilador poder ser gerado por um outro programa, entretanto, o número de nomes pode exceder em muito o da maioria dos programas de uso humano que sejam gerados. Se presumirmos esse caso, tamanhos maiores de tabelas de símbolos são preferíveis.

Uma boa atenção tem sido dada à questão de se projetar uma função de hash que seja fácil de computar para cadeias de caracteres, e tal que distribua igualmente quase igualmente cadeias típicas em listas.

Uma abordagem adequada ao cômputo das funções de hash está a ser descrita da seguinte forma:

1. Determinar um inteiro positivo i a partir dos caracteres $c_1, c_2, \ldots, c_n$ na cadeia s. A ordem seleção de quaisquer cinco ou seis caracteres é tipicamente suportada pela linguagem de implementação. Pascal provê, em sua função ord com esse propósito. Ord(c) retorna imediatamente um caráter para um inteiro. Se uma operação aritmética for realizada sobre o inteiro

2. Como de costume, determine o deslocamento do primeiro caractere não nulo na lista, isto é, um número entre 0 e 25 ou −1. Simplesmente atribua o valor 0 para qualquer caractere não nulo que ocorra depois do primeiro. Para continuar o exemplo, se a lista de caracteres for formada por consoantes, 211 será igual a 200 na Fig. 7.34.

As funções de *hash* que examinamos podem ser caracterizadas como sendo muito irregulares. Uma pequena alteração em uma chave pode produzir um endereço completamente diferente. Isso é importante para algumas características das extremidades de um nome de cadeia. Recomendamos que a entrada para o compilador possa ser lida de um arquivo-texto, por um programa e pode ser, essencialmente, de uma forma estabelecida escolhida para evitar conflitos com os nomes que uma pessoa ou outro programa pode querer utilizar. As pessoas costumam "agrupar" os nomes de tal forma que escolhas como

e assim por diante.

Uma técnica simples para computar h para cadeias mais longas é combinar o valor de h para os caracteres iniciais de uma cadeia com o valor de h para o restante da cadeia. Em outras palavras, se $h(s[0], s[1], \ldots, s[n])$ for igual a $h(s[0], h(s[1], \ldots, s[n]))$, então h é chamada de função de hash recursiva. Uma vez que a função de hash que propusemos é recursiva, é fácil escrever uma função recursiva para computar h. Entretanto, como vimos em capítulos anteriores, as funções recursivas podem ser ineficientes. Uma técnica alternativa consiste em calcular os valores de h para cadeias de comprimento crescente, começando com cadeias de comprimento 0 e terminando com cadeias de comprimento igual ao comprimento da cadeia de entrada. Isso pode ser feito facilmente em uma única passagem pela cadeia de entrada, da esquerda para a direita. Essa técnica é ilustrada pela função `hash` a seguir:

```
int hash(char *s)
{
    int h = 0;
    while (*s != '\0')
        h = 65599 * h + *s++;
    return h;
}
```

Fig. 7.35. Um exemplo de função de hash recursiva, escrita em C

Exemplo 7.10. Para resultados ótimos, o tamanho da tabela de hash e da entrada esperada precisam ser levados em consideração quando a função de hash for projetada. Para exemplificar, desejamos que os valores de hash para os nomes que ocorrem mais frequentemente numa linguagem sejam distintos. Se as palavras-chave forem incluídas na tabela de símbolos, estamos propensos a garantir que os nomes que ocorrem mais frequentemente, apesar de não serem palavras-chave, também sejam distintos. Em um dos exemplos de programa C, o nome `argv` ocorre frequentemente três vezes num dado programa.

Uma forma de se testar uma função de hash é examinar o número de colisões que ocorrem na entrada de um arquivo-texto. Consiste em calcular h para cada uma das n cadeias existentes, isto é, para $0 \le i < n$. Uma medida de quão uniformes são as cadeias é dada pela soma dos quadrados das listas de sinônimos:

$$\sum_i (b_i + 1)/2 \qquad (7.2)$$

A justificação intuitiva para essa forma estranha é que precisamos examinar b_i elementos de lista para encontrar a b_i-ésima entrada na lista. Para encontrar o segundo elemento, precisamos de mais uma comparação, e assim por diante. A soma de $1, 2, \ldots, b_i$ é $b_i(b_i + 1)/2$.

Do Exercício 7.14 o valor de (7.2) para uma função de hash que distribua cadeias aleatoriamente através dos m locais é dado por

$$n/2(m + n + 2m - 1) \qquad (7.3)$$

A média de 1 000 000 (7.2) é 1 767 e plotada na Fig. 7.36 para várias funções de hash aplicadas a três arquivos. Os arquivos são:

1. Os nomes e valores-chave que ocorrem mais frequentemente numa amostra de programas C
2. Como em 1, mas com os 100 nomes e palavras-chave que ocorrem mais frequentemente
3. Como em 1, mas com os 500 nomes e palavras-chave que ocorrem mais frequentemente
4. 952 nomes externos do kernel do sistema operacional UNIX
5. 677 nomes num programa C gerado por YACC (Johnson [1978b])
6. 915 endereços de caracteres gerados randomicamente
7. 614 palavras da Seção 3.1 deste livro, no original em inglês
8. 1201 palavras em inglês comuns, ordenadas como preferências (Ivie [1966])
9. Os 8192 nomes [illegible]

A função de hash usada é como na Fig. 7.35. As funções denominadas `x31`, onde `x` é uma constante, foram computadas usando `unsigned`, onde `h` retém o valor atualizado correspondente a cada caractere. A função `middle` é obtida a partir dos quatro caracteres centrais da cadeia (ou pelos dois caracteres centrais, se a cadeia for curta). Uma função `quadruple` agrupa os valores dos caracteres consecutivos em um único valor e os adiciona

Representando a Informação de Escopo

As entradas da tabela de símbolos são para declarações ou nomes. Quando uma ocorrência de um nome no texto-fonte é pesquisada na tabela de símbolos, a entrada para a declaração apropriada, daquele nome deve ser retornada. As regras de escopo da linguagem fonte determinam a declaração que é apropriada.

Uma abordagem simplista seria ter uma tabela de símbolos separada para cada escopo. Como exemplo, a tabela de símbolos para um bloco poderia conter um escopo específico, um campo de comprimento do registro de [illegible] caracteres. As informações para os nomes não locais de um procedimento são encontradas nas respectivas tabelas de símbolos para os procedimentos envolventes seguindo-se os registros de escopo da lista ligada. De modo equivalente, as informações sobre os nomes locais para um procedimento podem ser armazenadas numa área de memória livre semelhante para o programa. Com essa abordagem a tabela de símbolos é integrada à representação intermediária de entrada.

As regras de escopo da maioria das linguagens mais recentes podem ser implementadas adaptando-se as estruturas de dados apresentadas anteriormente neste capítulo. Comentaremos os nomes locais de um procedimento fornecendo a cada procedimento um único número. Os blocos também precisam ser numerados se a linguagem for estruturada em

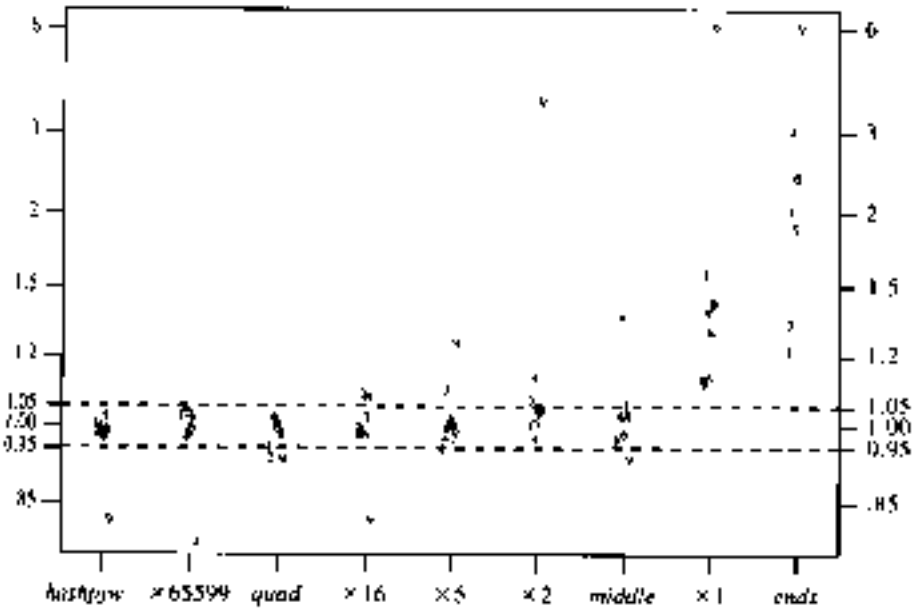


Fig. 7.36. Desempenho relativo de funções de hash para tabelas de tamanho 211

níveis. O número de cada procedimento pode ser exemplificado de uma forma dirigida pela sintaxe para incluir ações semânticas que incrementam um contador de cada procedimento. O número de procedimentos chamado para todos os nomes locais exclusivos exemplifica o custo para representação de nome local na tabela de símbolos e quantas vezes cada nome é processado. Os procedimentos semânticos também podem ser ativados para incluir outras ações, tais como aquelas descritas acima, e o número de procedimentos chamados pode ser reduzido. O processo é relativamente apurado, na medida em que pode ser ativado após a entrada ter sido posta nos registros na tabela de símbolos.

Quando precisamos percorrer uma tabela de símbolos, um método muito conveniente é usar uma lista encadeada que segue, para cada entrada, todas as referências do nome na entrada na tabela. Isso requer uma entrada adicional na tabela de símbolos (ou um ponteiro) para cada entrada processada. As regras de escopo de uma linguagem determinam quais entradas podem ser implementadas em termos das seguintes operações sobre um nome:

pushscope	cria uma nova entrada mais recentemente criada
enter(n)	constrói uma nova entrada
popscope	remove a entrada mais recentemente criada

As entradas "removidas" precisam ser preservadas, não apenas removidas da tabela de símbolos ativa. Para compilador de uma passagem, as informações sobre um escopo e um símbolo, digamos, no corpo de um procedimento, não são necessárias em tempo de compilação após o corpo do procedimento ter sido processado. Entretanto, podem ser necessárias em tempo de execução, particularmente se um sistema de depuração for implementado. Neste caso as alterações na tabela de símbolos precisam ser acrescentadas ao código gerado para manipulação pelo código de ligações ou pelo sistema de depuração em tempo de execução. Veja também o tratamento dos nomes de campos em registros nas Seções 8.2 e 8.3.

Cada uma das estruturas de dados descritas nesta seção — listas encadeadas — pode ser manipulada de forma adequada para implementação nessas operações. Quando uma lista encadeada é construída em uma área de registros descartáveis, como nesta seção, frequentemente pode ser implementada através da inserção de entradas numa das extremidades da tabela de tal forma que a ordem das entradas na lista seja a mesma ordem de inserção. Um esquema bastante econômico para a extremidade em direção ao início do vetor, encontra a entrada mais recente de cada nome. A situação é similar numa lista ligada, como ilustrado na Fig. 7.37. O apontador *scope* aponta para a entrada mais recentemente criada na lista. A implementação de *enter* é trivial: consiste em criar uma entrada e colocá-la à frente da lista. A implementação de *pushscope* é feita exatamente da mesma forma, apontando-se a lista correspondente para *scope* e seguindo-se os elos. A implementação de *popscope* é, exceto por uma dificuldade, igualmente simples. Na Fig. 7.37, a entrada para a declaração no bloco *B* — anulado no bloco *B* — aparece imediatamente à frente da lista do que a entrada para a declaração em *A*.

Para a operação *delete* é necessário que a entrada para as declarações do procedimento mais recente anulado apareça à frente da lista. Isso consegue-se mais facilmente mantendo-se um elo de precedência no próprio registro de cada entrada — se controlarmos o processo para cada procedimento, todas as entradas a serem removidas da tabela de símbolos aparecerão imediatamente à esquerda do procedimento.

Uma tabela de escopos consiste em listas as quais se tem acesso através de um apontador. Como um nome sempre tem o mesmo comprimento de hash calculado para a mesma lista, as listas individuais são mantidas como na Fig. 7.4(c). Entretanto, para implementar *popscope*, onde apenas um nome é removido de toda a tabela hash, procurando por listas com todas as entradas a serem removidas. A abordagem a seguir pode ser usada. Suponhamos que cada entrada possua dois elos:

1. um elo de hash que encadeia a entrada com as demais que contêm um mesmo valor de hash
2. um elo de escopo que encadeia todas as entradas que pertencem ao mesmo escopo

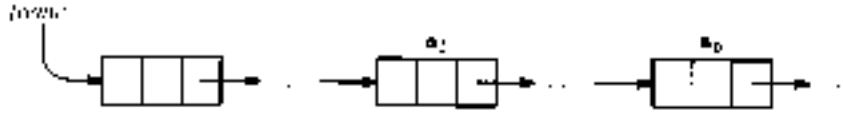


Fig. 7.37. A estrutura de escopo para [illegible] da Lista da Fig. 7.4

Se o elo de escopo [illegible] mantido no [illegible] ao se remover uma entrada numa tabela de *hash*, a cadeia formada pelos elos de escopo constituirá se uma [illegible] tabela de símbolos separada (não *hash*) para o escopo em questão.

A remoção de entradas da tabela de *hash* precisa ser realizada com cuidado, porque a remoção de uma entrada afeta a anterior na lista. Relembremos que removemos a *i*-ésima entrada fazendo com que a (*i* – 1)-ésima entrada aponte para a (*i* + 1)-ésima. Usar simplesmente os elos de escopo para encontrar a *i*-ésima entrada não é o bastante. A (*i* – 1)-ésima entrada [illegible] pode ser encontrada se as listas de *hash* formarem uma lista circular ligada, na qual a última entrada aponta de volta para [illegible]. Alternativamente, poderíamos usar uma pilha para controlar as listas que contenham entradas a serem removidas. Um marcador é colocado na pilha quando um novo procedimento é esquadrinhado. Acima do marcador estão os números das listas que contêm entradas para nomes declarados nesse procedimento. Ao terminarmos de processar o procedimento, os números de lista podem ser removidos da pilha até que o marcador para o procedimento seja encontrado. Um outro esquema é discutido no Exercício 7.11.

7.7 FACILIDADES DE LINGUAGEM PARA A ALOCAÇÃO DINÂMICA DE MEMÓRIA

Nesta seção, descrevemos brevemente as facilidades providenciadas por algumas linguagens para a alocação dinâmica de memória para dados sob controle do programa. O armazenamento para tais dados é usualmente obtido a partir do *heap*. Os dados alocados são frequentemente retidos até que sejam explicitamente liberados. A alocação em si pode ser explícita ou implícita. Em Pascal, por exemplo, a alocação explícita é realizada usando-se o procedimento padrão [illegible]. A execução de [illegible] reserva memória para o tipo de objeto apontado por [illegible] e [illegible] fica apontando para o objeto recém-alocado. A liberação é feita chamando-se o procedimento [illegible] na maioria das implementações de Pascal.

A alocação implícita ocorre quando a avaliação de uma expressão resulta na obtenção de memória para abrigar o valor de uma expressão. Lisp, por exemplo, aloca uma célula numa lista quando [illegible] é usado; as células que não podem ser mais atingidas são automaticamente reclamadas de volta. Snobol permite que o comprimento de uma cadeia varie em tempo de execução e gerencia o espaço necessário para abrigar uma cadeia numa área de comprimento do tipo [illegible].

Exemplo 7.11. O programa Pascal na Fig. 7.38 constrói a lista ligada mostrada na Fig. 7.39 e imprime os inteiros armazenados nas células.

```
( 1)  [illegible]
( 2)  [illegible]
( 3)  [illegible]
( 4)  [illegible]
( 5)  [illegible]
( 6)  [illegible]
( 7)  [illegible]
( 8)  [illegible]
( 9)  [illegible]
(10)  [illegible]
(11)  [illegible]
(12)  [illegible]
(13)  [illegible]
(14)  [illegible]
(15)  [illegible]
(16)  [illegible]
(17)  [illegible]
(18)  [illegible]
(19)  [illegible]
(20)  [illegible]
```

Fig. 7.38. Alocação dinâmica de células usando [illegible] em Pascal.

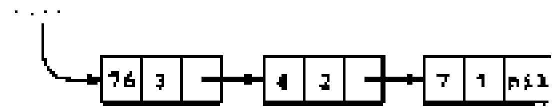


Fig. 7.39. Lista ligada construída pelo programa da Fig. 7.38.

Sua saída é

[illegible]

Quando a execução do programa começa à linha 15, o armazenamento do apontador [illegible] está no registro de ativação para todo o programa. A cada vez que o controle atinge

[illegible]

a chamada [illegible] resulta numa célula sendo alocada em algum lugar do *heap*; [illegible] se refere a essa célula na atribuição à linha 11.

Note-se, da saída do programa, que quando o controle retorna para o programa principal, proveniente de [illegible], as células alocadas estão acessíveis. Em outras palavras, as células alocadas, utilizando-se [illegible] durante uma ativação de [illegible], são retidas quando o controle retorna para o programa principal, proveniente da ativação [illegible].

Lixo

A memória alocada dinamicamente pode se tornar inatingível. A memória que um programa reserva mas não se refere é chamada de *lixo*[1]. Na Fig. 7.38, suponhamos que [illegible] fosse atribuído a [illegible] entre as linhas 16 e 17.

```
(16)  [illegible]
      [illegible]
(17)  [illegible]
```

A célula mais à esquerda na Fig. 7.39 [illegible] agora contém um valor de apontador [illegible] ao invés de um apontador para a célula ao meio. Quando o apontador para a célula do meio está perdido, as células do meio e a mais à direita se tornam lixo.

Lisp realiza a *coleta de lixo*, um processo discutido na próxima seção, que reclama a memória inacessível. Pascal e C não têm coleta de lixo, deixando para o programa liberar explicitamente a memória que já não é mais desejada. Nessas linguagens, a memória liberada pode ser reusada, mas o lixo permanece até o programa terminar.

Referências Ocas

Uma complicação adicional pode emergir com a liberação explícita: as referências ocas podem ocorrer. Como mencionado na Seção 7.1, uma referência oca ocorre quando se referencia uma memória que já foi liberada. Por exemplo, consideremos o efeito de se executar [illegible] entre as linhas 16 e 17 na Fig. 7.38.

```
(16)  [illegible]
      [illegible]
(17)  [illegible]
```

A chamada a [illegible] libera a célula seguinte àquela apontada por [illegible], como mostrado na Fig. 7.40. Entretanto, [illegible]

[1] [illegible]

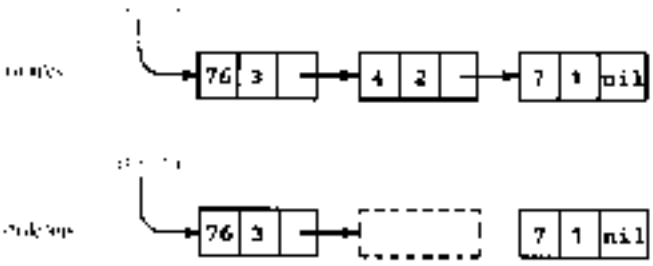


Fig. 7.40 Criação de referências cruzadas de lixo

... não for multiplicado e desta forma, será um apontador nos para a própria lista.

As técnicas vistas até o momento são genéricas e tratam os registros de forma uniforme, liberando-os sempre que a última referência deixa de existir. Se a última referência ocorrer antes da liberação, o registro não será liberado e ficará inacessível, ocupando espaço de memória até o término do programa. Esse problema é conhecido como *lixo*.

7.8 TÉCNICAS DE ALOCAÇÃO DINÂMICA DE MEMÓRIA

As técnicas necessárias à implementação da alocação dinâmica de memória consistem de como a memória é liberada. Se a liberação for explícita, o usuário, ao supor em um momento que não precisará mais pela determinação de quando uma variável não é mais necessitada. Haverá também a possibilidade por parte do compilador se a liberação for feita explicitamente pelo programador. Considere após a liberação explícita ponteiro.

Alocação Explícita de Blocos de Tamanho Fixo*

A forma mais simples de alocação dinâmica envolve blocos de tamanho fixo. Através da ligação dos blocos numa lista, como mostra a Fig. 7.41, a reserva e liberação podem ser feitas rapidamente, com pouca ou nenhuma sobrecarga de memória.

Suponhamos que os blocos devam ser retirados de uma área contígua de memória. A inicialização começa reservando uma parte de cada bloco como um ponteiro para o bloco seguinte. O apontador *primeiro disponível* aponta para o primeiro bloco. A alocação consiste em substituir um bloco da lista e a liberação em colocar o bloco de volta na lista.

As rotinas do compilador que gerenciam os blocos não possuem saber o tipo de objeto de dados que será guardado no bloco por parte do programa do usuário. Podemos tratar cada bloco como um registro variante, com o código do compilador encarregando cada bloco como consistindo em um ponteiro para o próximo bloco e o programa do usuário, que pode ser, também, um tipo de algum outro tipo. Por conseguinte, não existe sobrecarga de espaço, porque o programa do usuário pode utilizar todos os blocos para suas próprias finalidades. Quando o bloco é devolvido, as rotinas do compilador usam parte do espaço do próprio bloco para ligá-lo à lista de blocos disponíveis, como mostrado na Fig. 7.41.

Alocação Explícita de Blocos de Tamanhos Variados**

Quando os blocos são alocados e liberados, a memória se torna *fragmentada*, isto é, o livre pode consistir em blocos que estejam alternadamente livres e ocupados, como na Fig. 7.42.

A situação mostrada na Fig. 7.42 pode ocorrer se um programa reservar cinco blocos e os liberar segundo o seu tamanho, por exemplo. A fragmentação não gera consequências se os blocos são de tamanho fixo, mas se são de tamanhos variados a situação da Fig. 7.42 é problemática, porque poderíamos não alocar um bloco maior do que qualquer um dos blocos livres, apesar de o espaço total disponível

Um método para se alocar blocos de tamanhos variados é a utilização de *primeiro disponível*.** Quando um bloco de tamanho é solicitado, pesquisamos pelo primeiro bloco livre que tenha tamanho . Esse bloco é então subdividido em um bloco usado de tamanho e um bloco livre de tamanho . Note se que a alocação envolve uma sobrecarga de tempo, porque é necessário se procurar por um bloco livre que seja suficientemente grande.

Quando um bloco é liberado, verificamos se o mesmo é adjacente a um bloco livre. Se possível, o bloco liberado é combinado com o bloco livre adjacente a ele de forma a criar um bloco livre maior. A combinação de blocos adjacentes livres em um bloco livre maior previne

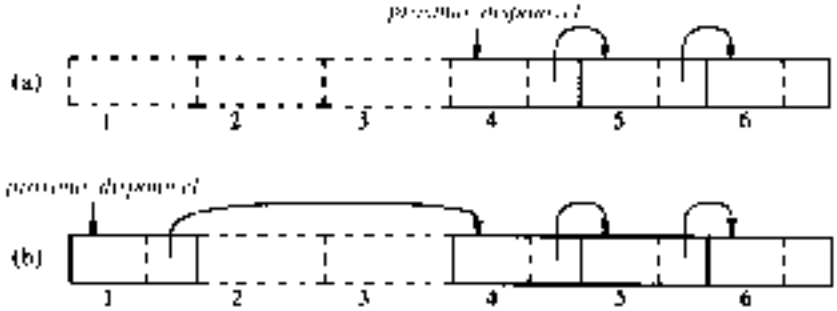


Fig. 7.41 Um bloco livre contém o endereço do próximo bloco livre

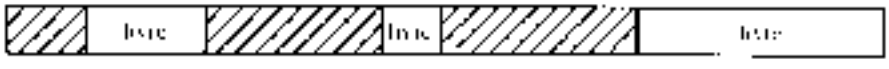


Fig. 7.42 Blocos livres e ocupados em uso

* Também conhecida como Alocação Estática. Nesta técnica, todos os blocos são de mesmo tamanho. Assim, todos os blocos são alocados com o mesmo tamanho, e todos os blocos livres são mantidos em uma lista. (N. do T.)

** Também conhecida como Alocação Dinâmica. (N. do T.)

que ocorra uma fragmentação posterior. Nessa situação, os detalhes sutis relacionados a quando os blocos são alocados, liberados e mantidos numa lista ou listas de disponibilidade. Existem também muitos compromissos entre tempo, espaço e a disponibilidade de grandes blocos. O leitor é remetido a Knuth [1973a] ou a Aho, Hopcroft e Ullman [1983] para uma discussão desses temas.

Liberação Implícita

A liberação implícita requer a cooperação entre o programa do usuário e o pacote de suporte em tempo de execução, porque este último necessita saber quando um bloco de memória não está mais em uso. Essa cooperação é implementada através da fixação do formato dos blocos de memória. Para a discussão presente, suponhamos que o formato de um bloco de memória seja como na Fig. 7.45.

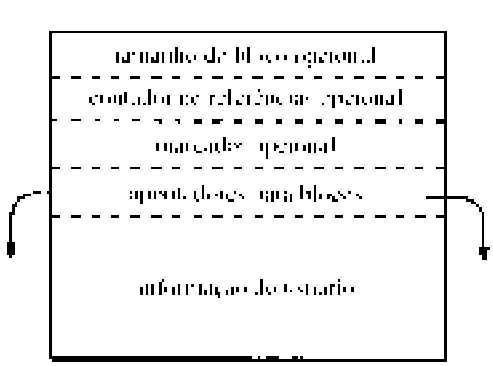


Fig. 7.45. Formato de um bloco.

O primeiro problema é o de reconhecer os limites de um bloco. Se o tamanho do bloco é fixo, informações de posição podem ser usadas. Por exemplo, se cada bloco ocupa 20 palavras, um novo bloco se inicia a cada 20 palavras. Em caso contrário, mantemos o seu tamanho numa área não-acessível associada ao bloco, de forma que possamos determinar onde começa o próximo.

O segundo problema é o de reconhecer se um bloco está em uso. Assumimos que um bloco esteja em uso se for possível que o programa do usuário se refira à informação nele contida. A referência pode ocorrer através de um apontador ou após se seguir uma sequência de apontadores e, por conseguinte, o compilador precisa conhecer a posição de todos os apontadores na memória. Usando-se o formato da Fig. 7.45, os apontadores são mantidos em posições fixas nos blocos. Talvez mais adequada seja a suposição de que a área do bloco reservada às informações do usuário não contenha quaisquer apontadores.

Duas abordagens podem ser usadas para a liberação implícita. Aqui, os métodos são esboçados; para maiores detalhes, ver Aho, Hopcroft e Ullman [1983].

1. *Contadores de referências.* Controlamos o número de blocos que apontam diretamente para o bloco presente. Se esse contador chegar a zero, o bloco pode ser desalocado, uma vez que não pode ser referenciado. Em outras palavras, se torna lixo e pode ser coletado. A manutenção de contadores de referências pode custar muito em tempo. A atribuição de apontador p := q leva a mudanças no contador de referências dos blocos apontados tanto por p quanto por q. O contador para o bloco apontado por p decresce de um, enquanto que o do bloco apontado por q aumenta de um. Os contadores de referências são melhor utilizados quando os apontadores de um bloco nunca aparecem em ciclos. Por exemplo, na Fig. 7.44, nenhum dos dois blocos está acessível por parte de outro e, dessa forma, ambos são lixo, mas cada um tem uma contagem de referências diferente de zero igual a um.

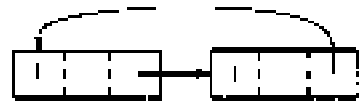


Fig. 7.44. Células que são lixo com contadores de referência diferentes de zero.

2. *Técnicas de marcação.* Um enfoque alternativo é o de suspender temporariamente a execução do programa do usuário e utilizar os apontadores congelados para determinar quais dos blocos estão em uso. Esse enfoque requer que todos os apontadores dentro do *heap* sejam conhecidos. Conceitualmente, "despejamos tinta" no *heap* através dos apontadores. Qualquer bloco que seja atingido pela tinta está em uso e o resto pode ser liberado. Em maiores detalhes, varremos o *heap*, marcando todos os blocos como *não usados*. Depois, seguimos os apontadores marcando como *usado* qualquer bloco que tenha sido atingido durante o processo. Uma varredura final do *heap* permite que todos os blocos que ainda estejam marcados como *não usados* sejam coletados.

Com blocos de tamanho variável, temos a possibilidade adicional de mover blocos de memória utilizados a partir de suas posições correntes.[?] Esse processo, chamado de *compactação*, move todos os blocos usados para uma extremidade do *heap*, de forma que toda a memória livre possa ser coletada num único grande bloco livre. A compactação também requer informações sobre os apontadores dentro dos blocos, porque quando um bloco é movido, todos os apontadores para o mesmo precisam ser ajustados de forma a refletir a nova localização. A vantagem está em que é eliminada a fragmentação da memória disponível ao usuário.

7.9 ALOCAÇÃO DE MEMÓRIA EM FORTRAN

Fortran foi projetada para permitir a alocação estática de memória, como estabelecido na Seção 7.3. No entanto, existem alguns temas, tais como o tratamento das declarações COMMON e EQUIVALENCE, que são razoavelmente específicos em Fortran. Um compilador Fortran pode criar um certo número de áreas de dados, isto é, blocos de memória nos quais os valores dos objetos podem ser armazenados. Em Fortran, há uma área de dados para cada procedimento, para cada bloco COMMON nomeado e para o bloco COMMON em branco, se for usado. A tabela de símbolos precisa registrar, para cada nome, a área de dados à qual pertence e o seu deslocamento dentro dessa área, isto é, a posição relativa ao início da mesma. O compilador terá que, eventualmente, decidir onde as áreas estarão em relação ao código executável e na relação entre si, mas esta escolha é arbitrária, uma vez que as áreas de dados são independentes.

O compilador precisa computar o comprimento de cada área. Para as áreas de dados dos procedimentos, um único contador basta, uma vez que seus tamanhos são conhecidos depois de cada procedimento ter sido processado. Para os blocos COMMON, um registro para cada bloco precisa ser mantido durante o processamento de todos os procedimentos, uma vez que cada procedimento usando um desses blocos pode ter a sua própria idéia a respeito de quão grande o bloco é e o tamanho efetivo será o maior dos tamanhos implicados pelos vários procedimentos. Se os procedimentos forem compilados separadamente, o editor de ligações precisa ser usado a fim de selecionar o tamanho do bloco COMMON para o maior de todos esses blocos, com o mesmo nome, dentre os blocos de código que estiverem sendo ligados.

Para cada área, o compilador cria um *mapa de memória*, que é uma descrição do seu conteúdo. Esse "mapa de memória" poderia consistir simplesmente em uma indicação, na entrada da tabela de símbolos para cada nome existente na área, do seu respectivo deslocamento dentro da mesma. Não precisamos ter, necessariamente, uma resposta fácil para a questão "quais são todos os nomes dessa área de dados?" Em Fortran

[?] [illegible]

[illegible] sabemos a resposta para a soma de todos os procedimentos [illegible] nomes declarados num procedimento que não sejam [illegible] estejam relacionados com nomes [illegible] estão na área de dados do procedimento. Os nomes [illegible] podem ser suas entradas [illegible] de símbolos ligadas a uma cadeia para cada bloco [illegible] classificada pela ordem de aparição dos nomes no bloco. De fato, na área em que os deslocamentos dos nomes na área de dados são postos [illegible] sempre são inicializados até que todo o procedimento tenha sido processado, os nomes em locais podem ser declarados antes de suas direções serem [illegible] necessário que seja [illegible] cada [illegible] de nomes [illegible].

Um programa, [illegible] visto como um programa principal, [illegible] sub-rotinas declaradas [illegible] nomes [illegible] *procedimentos*. Cada procedimento [illegible] pode ter um escopo que consiste em um procedimento [illegible] seus filhos imediatos. Podemos gerar código-objeto para cada procedimento isoladamente, fazendo referências [illegible] nomes não declarados localmente. Quando [illegible] procedimento [illegible] nomes não declarados localmente, [illegible] tabela de símbolos para encontrar [illegible] nome. Se [illegible] não for encontrado [illegible] nome, [illegible] tabela de símbolos do procedimento [illegible] e assim por diante, até [illegible] nome ser encontrado [illegible] tabela de símbolos do programa principal. Se [illegible] não for encontrado [illegible] nome, [illegible] um erro [illegible] nome não declarado.

Essas informações precisam ser verdadeiras [illegible] quando [illegible] código-objeto [illegible] executado, mas precisam ser preservadas [illegible] procedimento [illegible] seja [illegible].

Dados em Áreas [illegible]

Criamos, para cada bloco, um registro fornecendo o primeiro e último nomes pertencentes [illegible] procedimento corrente. Esses dados [illegible] [illegible] [illegible] de bloco [illegible]. Ao processar uma declaração como

[illegible]

[illegible]

1. A tabela para nomes de blocos [illegible] criamos um registro para [illegible] já [illegible].
2. Se [illegible] da tabela de símbolos para [illegible] estabelecer [illegible] um apontador para a entrada da tabela de símbolos de [illegible] duplicando [illegible] nomes são membros de um bloco [illegible] de nome [illegible].
3. Se [illegible] pode ser criado um registro para [illegible] estabelecer um apontador [illegible] para a entrada de [illegible] na tabela de símbolos, indicando o propósito nome definido nesse bloco [illegible]. Em seguida, ligar a entrada da tabela de símbolos de [illegible] [illegible] da tabela de [illegible], usando um campo da tabela de símbolos reservado para ligar os membros de um bloco [illegible]. Finalmente, estabelecer um apontador no registro para [illegible] para a entrada da tabela de símbolos de [illegible] indicando o último membro encontrado para aquele bloco.
4. Se [illegible], [illegible] [illegible] para [illegible] simplesmente ligar [illegible] ao final da lista de nomes para [illegible]. Apontadores para a [illegible] da lista para [illegible] ligando-o no registro para [illegible], e [illegible] adequadamente.

Após o procedimento [illegible] processar, [illegible] algoritmo de equivalência, [illegible] descrito em breve. Podemos descobrir que [illegible] nomes pertencentes a um bloco [illegible] foram [illegible] equivalentes a nomes que estão em blocos [illegible]. Iremos deduzir [illegible] de fato necessário ligar [illegible] à cadeia para esse bloco [illegible]. Em cada entrada da tabela de símbolos para [illegible] ligado indicando que [illegible] equivalente a algo. Uma estrutura de dados [illegible], daí a posição de [illegible] relativa a algum nome [illegible] declarado como estando em um bloco [illegible].

Após [illegible] as operações de equivalência, podemos criar um mapa de memória para cada bloco [illegible], esquadrinhando a lista de nomes para aquele bloco. Inicializamos um contador em zero e, para cada nome na lista, fazemos seu deslocamento igual ao valor corrente [illegible]. Em seguida, adicionamos o número de unidades de memória ocupadas pelo objeto de dados declarado pelo nome. Os registros para os blocos [illegible] podem [illegible] removidos e o espaço reusado pelo próximo procedimento.

Se um nome [illegible] um bloco [illegible] equivalente a um nome que não esteja num bloco, precisamos determinar o deslocamento máximo, a partir do início de [illegible], de qualquer pedaço de memória necessário por qualquer nome equivalente [illegible]. Por exemplo, se [illegible] um real, [illegible] e [illegible] um [illegible] [illegible] [illegible] ligar 44 palavras antes de [illegible] [illegible] 55 palavras após [illegible], como mostrado na Fig. 7.45. A [illegible] de [illegible] deslocamento para bloco [illegible] [illegible] é considerado, independentemente de [illegible] tenhamos [illegible] para um que [illegible] seja. Entretanto, [illegible] área de dados para o bloco [illegible] precisa estar distante o bastante de [illegible] nomes [illegible] [illegible]. Por conseguinte, registramos [illegible] a partir do início do bloco [illegible] de qualquer palavra usada por um nome equivalente a um membro daquele bloco. Na Fig. 7.45, esse valor [illegible] ser, pelo menos, o deslocamento de [illegible] mais 55. Verificamos também se [illegible] não se estende para antes do início da área de dados; isto é, o deslocamento de [illegible] precisa ser pelo menos 44. Em [illegible] temos uma [illegible] que [illegible] produzir uma necessidade de [illegible].

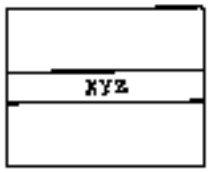


Fig. 7.45. Relação entre [illegible]

Um Algoritmo Simples de Equivalência

Os primeiros algoritmos para processar conjuntos de equivalência apareceram nos sistemas operacionais de [illegible] e de [illegible] computadores. Esses algoritmos podem ser [illegible] complexos, especialmente quando novas relações entre [illegible] são [illegible]. Vamos considerar, [illegible] um caso [illegible] de uma [illegible] de montagem, onde as [illegible] enumerados [illegible] em declarações da forma

[illegible]

onde [illegible] são nomes de localizações. Esse comando faz [illegible] a localização que está [illegible] unidades de memória antes da localização de [illegible].

Uma sequência de enumerados [illegible] agrupada em *conjuntos de equivalência*, [illegible] cujas posições relativas entre [illegible] são definidas pelos próprios enumerados [illegible]. Por exemplo, a sequência de enumerados

[illegible]

agrupa os nomes nos conjuntos [illegible] onde [illegible] declaram a mesma localização [illegible] estão [illegible] após [illegible] estão [illegible] após [illegible].

Para computar os conjuntos de equivalência, criamos uma árvore para cada conjunto. Cada nó da árvore representa um nome e contém o deslocamento daquele nome relativo ao nome que está no pai desse nó. Desconsideramos o nome que está à raiz da árvore de [illegible]. A posição de qualquer nome, relativa ao líder, pode ser computada se

guindo-se o percurso a partir do nó para a palavra-nome e adicionando-se os deslocamentos ao longo do caminho, até a raiz.

Exemplo 7.12. O conjunto de equivalência {A, B, C, D} descrito acima poderia ser representado pela árvore mostrada na Fig. 7.46. *D* é a raiz e podemos descobrir que *A* está localizado 10 posições antes de *D*, porque a soma dos deslocamentos ao longo do percurso que vai de *A* até *D* é 100 + (−110) = −10. □

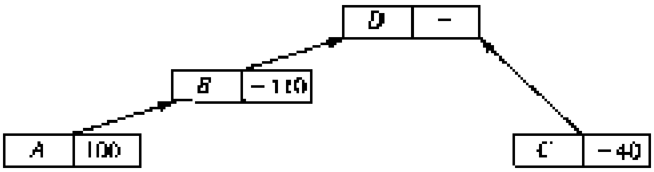


Fig. 7.46. Árvore representando o conjunto de equivalência.

Vamos agora fornecer um algoritmo para a construção de árvores para conjuntos de equivalência. Os campos relevantes nas entradas da tabela de símbolos são:

1. *pai*, apontando para a entrada da tabela de símbolos para o pai, nulo se o nome estiver na raiz (ou não equiparado a qualquer outro).
2. *deslocamento*, fornecendo o deslocamento do nome em relação ao nome do pai.

O algoritmo que forneceremos assume que qualquer nome poderia ser o líder de um conjunto de equivalência. Na prática, numa linguagem de montagem, um dos nomes no conjunto teria uma localização efetiva dada por uma pseudo-operação e esse nome seria tornado o líder. Confiamos em que o leitor possa ver como modificar o algoritmo e fazer um nome particular de líder.

Algoritmo 7.1. Construção de árvores de equivalência.

Entrada. Uma lista de enunciados definindo equivalências da forma

$$\text{EQU } A, B + dist$$

Saída. Uma coleção de árvores tal que, para qualquer nome mencionado na lista de equivalências, possamos determinar a posição do nome relativa ao líder, seguindo o percurso deste até aquele nome até a raiz e somando os *deslocamentos* encontrados ao longo do caminho.

Método. Repetimos os passos da Fig. 7.47 para cada enunciado de equivalência EQU *A*, *B* + *dist*, em turnos. A justificativa para a fórmula da linha (12) para o deslocamento do líder de *A* em relação ao líder de *B* é como segue. A localização de *A*, digamos l_A, é igual a *c* mais a localização do líder de *A*, digamos m_A. A localização de *B*, digamos l_B, é igual a *d* mais a localização do líder de *B*, digamos m_B. Mas $l_A = l_B + dist$, logo $c + m_A = d + m_B + dist$. Por conseguinte, $m_A - m_B$ é igual a $d - c + dist$. □

Exemplo 7.13. Se processarmos

EQU A, B+100
EQU C, D−40

obteremos a configuração mostrada na Fig. 7.48, com os deslocamentos −100 no nó para *B* e de −40 no de *D* para *C*. Ao processarmos

EQU A, C+30

encontraremos que *p* aponta para *B* após o laço enquanto à linha (3) e que *q* aponta para *D* após o laço enquanto à linha (6). Temos igualmente que *c* = 100 e que *d* = −40. Por conseguinte, à linha (11) fazemos *D* o pai de *B* e estabelecemos o campo *deslocamento* para *B* com o valor −110, que é o resultado de (−40) − (100) + 30. □

```
início
(1)   sejam p e q apontadores para os nós de A e B, respectivamente;
(2)   c := 0; d := 0; /* c e d computam os deslocamentos de A e B
          a partir dos líderes de seus respectivos conjuntos */
(3)   enquanto pai(p) ≠ nulo faça início
(4)       c := c + deslocamento(p);
(5)       p := pai(p)
      fim; /* move p para o líder de A, acumulando os
                 deslocamentos à medida que caminhamos */
(6)   enquanto pai(q) ≠ nulo faça início
(7)       d := d + deslocamento(q);
(8)       q := pai(q)
      fim; /* fazer o mesmo para B */
(9)   se p = q então /* A e B já estão equiparados */
(10)      se c − d ≠ dist então erro;
              /* foram atribuídas duas posições relativas diferentes
                 a A e B */
      senão início /* combinar os conjuntos de A e B */
(11)      pai(p) := q; /* fazer o líder de A um filho do líder de B */
(12)      deslocamento(p) := d − c + dist
      fim
fim
```

Fig. 7.47. Algoritmo de equivalência.

O Algoritmo 7.1 poderia levar um tempo proporcional a *n*² para processar *n* equivalências, uma vez que, no pior caso, os percursos seguidos pelos laços às linhas (3) e (6) poderiam incluir cada nó de suas respectivas árvores. As equiparações tomadas da equivalência requerem apenas uma pequena fração do tempo gasto durante a compilação e, dessa forma, *n*² passos não é um preço proibitivo e um algoritmo mais complexo do que aquele da Fig. 7.47 não é provavelmente justificável. No entanto, acontece que existem duas coisas fáceis que podemos fazer para tornar o tempo que o Algoritmo 7.1 gasta apenas linear em relação ao número de equivalências processadas. Conquanto não seja provável que os conjuntos de equivalência sejam grandes o suficiente, em média, de forma que esses aprimoramentos precisem ser realizados e implantados, é importante notar que a equiparação serve como um paradigma para um certo número de processos importantes envolvendo a "combinação de conjuntos". Por exemplo, um número de algoritmos eficientes para a análise do fluxo de dados depende de algoritmos de equivalência rápidos; o leitor interessado é remetido às notas bibliográficas do Capítulo 10.

O primeiro aprimoramento que podemos fazer é manter uma contagem, para cada líder, do número de nós em sua árvore. Então, às linhas (11) e (12), em lugar de ligar arbitrariamente o líder de *A* ao líder de *B*, ligue-se o que tiver a menor contagem ao outro. Isto assegura que as árvores cresçam achatadas, de forma que os percursos serão curtos. É deixado como exercício para o leitor comprovar que *n* equivalências realizadas dessa forma não podem produzir percursos mais longos do que log *n* nós.

A segunda idéia é conhecida como compressão de percursos. Ao seguir um percurso até a raiz, nos laços às linhas (3) e (6), faça todos

```
início
    h := deslocamento(p);
    para i := k−2 de −1 até 1 faça início
        pai(r_i) := q;
        h := h + deslocamento(r_i);
        deslocamento(r_i) := h
    fim
fim
```

Fig. 7.48. Ajuste dos deslocamentos.

...são encontrados ...líderes de líder, e ponteiros para ... Esses ponteiros ... para ... encontrar todos os nós ... encontrar ... para ... ou para ... líder. Esses ponteiros, assim como os descritores ... líderes ... através dos ponteiros ... Fig. 7.48.

Um Algoritmo de Equivalência para Fortran

Existem diversas ligações que podem ser feitas entre os algoritmos ... Fortran. Primeiro, precisamos ... conjunto de equivalência está em ... que ... fazer através do registro para cada líder, informando ... quantos nomes no seu conjunto ... está em ... bloco.

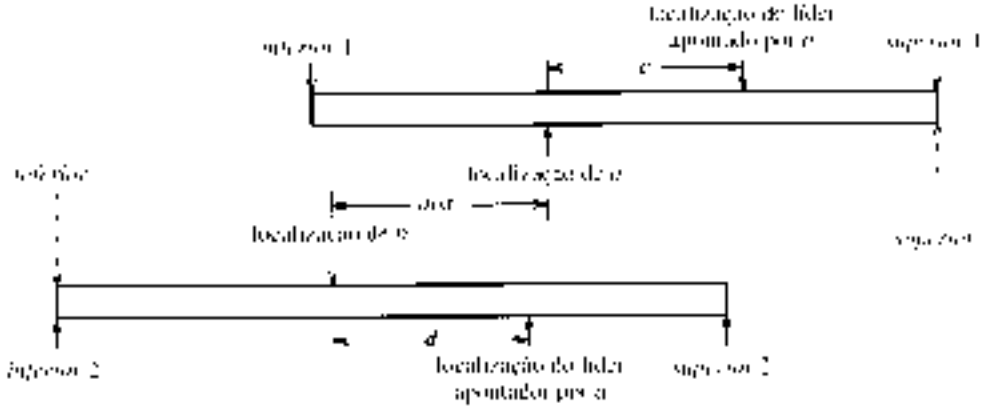
Segundo, numa linguagem de montagem ... membro de um conjunto de equivalência irá materializar todos os conjuntos pertencentes ... um conjunto, permitindo, por conseguinte, que todos os ... demarcados por todos os elementos daquele conjunto ... em relação àquela localização. Um Fortran ... o compilador determinar as localizações de memória ... para um conjunto de equivalência que ... bloco ... pode ser considerado como "buracos" ... compilador ... a posição de todos os conjuntos ... apropriada. Para realizar essa tarefa ... o conjunto de equivalência ... número de localizações ... nomes ... conjunto ... Para ... ... campos, além ... líder, dois campos ... gerais ... relativos ao líder das localizações mais baixa e mais alta ... por qualquer membro ... Finalmente, existem problemas ... que os ... podem ser ... as localizações ... podem ser ... localizações em outros ...

Uma vez que existem três campos ... e um apontador ... bloco ... que precisam ser associados a cada líder, ... reservar espaço para esses campos em todos os ... líderes da tabela de símbolos. Uma ... ação é usar ... Algoritmo 7.1 de forma a ... os ... líder para uma ... tabela com três campos ... Na medida em que essa tabela ... de símbolos ... separadas, pode ... registrar ... tabela ... endereço. Alternativamente, a tabela de símbolos pode ter um ... nodo, ... um nome e ... ponteiro ... líder. Se o espaço ... realmente a ... um algoritmo alternativo ... esta tabela, ... as ... de um pequeno ... de perguntas ... e discutido ... exercícios.

Vamos ... os dados que devem substituir as linhas 11 e 12 da Fig. 7.47. A situação na qual dois conjuntos de equivalência ... líderes são apontados por ... precisam ser combinados ... Fig. 7.49(a). A estrutura de dados que ... os dois conjuntos ... na Fig. 7.49(b). Primeiro, precisamos ... que não existam ... membros ... dos conjuntos de equivalência ... estejam em ... Mesmo que ... estejam no mesmo ... Fortran proíbe que sejam equiparados. Se qualquer ... ... pertencer a membros de ambos os conjuntos de equivalência, o conjunto combinado possui um apontador ... para o registro daquele bloco ... O código que faz essa verificação, assim ... é ... pelo ... líder do conjunto combinado, é mostrado na Fig. 7.50. No lugar das linhas 11 e 12 da Fig. 7.47, precisamos também ... a extensão do conjunto de equivalência combinado. A Fig. 7.49(a) indica as fórmulas para os novos ... de ... e ... relativos ao líder apontado por y.

```
início
    ... := ... + ...
    ... := ... + ...
    se ... ≠ null e ... ≠ null então ... := null; ... := 2 * ... - ... - ... - ... + 1
    senão se ... ≠ null então
        ... := ... + ...
fim
```

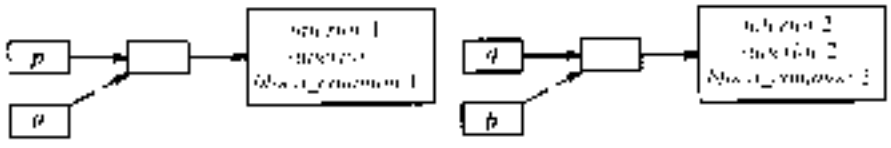
Fig. 7.50 ...



$$deslocx = \text{número de elementos } 2 \text{ antes do } 1 \ldots = deslocx - d$$

$$deslocy = \text{número de elementos } 2, \text{ conjunto } 1 = \ldots + deslocy + d$$

(a) posições relativas dos conjuntos de equivalência.



(b) estrutura de dados

Fig. 7.49. Conjuntos de equivalência ...

Por conseguinte, precisamos fazer

início
  sobrescrever posição := mínimo *sobrescrever posição* e *início bloco ativo*;
  sobrescrever posição := máximo *sobrescrever posição* e *sobrescrever posição*
fim

Essas instruções são seguidas pelas linhas (11) e (12) da Fig. 7.47, para efetuar a combinação dos dois conjuntos de equivalência.

Dois últimos detalhes precisam ser cobertos para fazer o Algoritmo 7.1 funcionar para Fortran. Em Fortran, podemos especificar blocos de memória de equivalência, posições em cima de uma *mesma* área de posições em outros blocos ou variáveis simples. O deslocamento do nome *a* a partir do líder significa o deslocamento da primeira localização de *a* a partir da primeira localização do líder. Se uma localização *x* do bloco *b* é equiparada, digamos, a *y*, então precisamos computar a posição de *x* relativa a *y* e inicializar para o valor negativo desse deslocamento na linha 2 da Fig. 7.49. Similarmente, *x* precisa ser inicializado para o negativo da posição de *y* relativa a *x*. As fórmulas da Seção 6.3 pressupõem o conhecimento do tamanho dos elementos dos arranjos, e não são suficientes para calcular os valores iniciais de *x* e *y*.

O último detalhe a ser coberto é o fato de Fortran permitir um comando *equivalence* que envolva uma das localizações, tal como

```
equivalence (a(10), b(20))
```

Isso significa que

```
a(1) está na mesma posição que b(11)
a(2) está na mesma posição que b(12)
...
```

Note-se que as exceções às equivalências nesta podem somente ocorrer com o líder de um conjunto com mais de um elemento. O registro com *deslocamento após o início do bloco* pode ser usado muitas vezes para "exceções de equivalência" de mais de um nome.

Mapeando Áreas de Dados

Podemos descrever agora as regras através das quais o espaço nas diversas áreas de dados é atribuído aos nomes pertencentes a cada rotina.

1. Para cada bloco *b* declarado na rotina, visitar todos os nomes declarados como pertencentes àquele bloco, na ordem de suas declarações. Usar as cadeias de nomes criadas na tabela de símbolos para esse propósito. Avaliar o número de palavras necessitado para cada nome, mantendo uma contagem do número de palavras alocadas, de forma que os deslocamentos possam ser computados para cada nome. Se um nome *x* for equiparado através de uma declaração de equivalência a outro nome, a extensão do seu conjunto de equivalência não importa, mas precisamos assegurar que o valor de *sobrescrever* para o líder de *x* não seja maior do que o tamanho do bloco. Consultar o valor sobrescrever para o líder, de forma a colocar um limite inferior na última palavra do bloco. Deixamos as fórmulas exatas para esses cálculos por conta do leitor.
2. Visitar todos os nomes para a rotina em qualquer ordem.
   a) Se um nome estiver em um bloco, não fazer nada. O espaço já foi reservado em (1).
   b) Se um nome não estiver em um bloco e não for equiparado, através de uma lista de equivalência, reservar o número adequado de palavras na área de dados para a rotina.
   c) Se um nome *x* for equiparado através de uma declaração de equivalência, encontrar o seu líder, digamos *y*. Se *y* ainda não foi atribuída uma posição na área de dados para a rotina, computar a posição de *x* sobrescrevendo àquela posição todos os valores menores encontrados no percurso de *y* até o início. Isso representa todos os compostos de equivalência de *x* e de *y*. Se *y* ainda não foi fornecida uma posição, ubique as previsões *sobrescrever* e *início* para *y* na área de dados, para o conjunto de equivalência. A posição de *y* nessas palavras está sobrescrevendo todos os valores menores encontrados. A posição de *x* pode ser calculada somando os deslocamentos constantes.

EXERCÍCIOS

7.1 Usando as regras de escopo de Pascal, determine as declarações que se aplicam a cada ocorrência dos nomes *x*, *y* e *w* na Fig. 7.50. A saída do programa consiste nos inteiros de 1 até 8.

```
program p(output);
var x, y, w: integer;
  procedure a;
  var x, y: integer;
    procedure b;
    var x, w: integer;
    begin
      x := 1; y := 2; w := 3;
      write(x, y, w)
    end;
  begin
    x := 4; y := 5; w := 6;
    write(x, y, w);
    b
  end;
begin
  x := 7; y := 8; w := 9;
  a
end.
```

Fig. 7.50. Programa Pascal com várias declarações para *x*.

7.2 Considere uma linguagem estruturada em blocos, na qual os nomes possam ser declarados como um inteiro ou real. Suponha que as expressões sejam representadas por um tipo especial que os inteiros entendidos separadamente, convertidos em constantes, comandos *while* e sequências de comandos. Assumindo que os inteiros sejam alocados em uma palavra e os reais em duas, forneça uma definição dirigida pela sintaxe baseada numa gramática razoável para as linguagens e blocos para determinar os endereços, a partir dos nomes, para as palavras de memória que passam a ser usadas por ocasião da entrada de um bloco. Você alocação case um número mínimo de palavras adequadas a qualquer execução de um bloco.

*7.3 Na Seção 7.4, afirmamos que o *dispose* poderia ser tratado corretamente se cada procedimento à profundidade *n* armazenasse *d*[*n*] no início de sua ativação e restaurasse *d*[*n*] no final. Prove, por indução no número de chamadas que cada procedimento executa, a exatidão desse fato.

7.4 Construa uma rotina de processamento, implementada substituindo literalmente cada chamada do procedimento pelo seu corpo. A Fig. 7.52 mostra o programa Pascal resultante. As duas primeiras linhas definem os nós de texto e de texto.

```
program p(output);
var x, y: integer;
  procedure a;
  var x, y: integer;
    procedure b;
    var x, w: integer;
    begin
      x := 1; y := 2; w := 3;
      write(x, y, w)
    end;
  begin
    x := 4; y := 5; w := 6;
    write(x, y, w);
    b
  end;
begin
  x := 7; y := 8; w := 9;
  a
end.
```

Fig. 7.52. Círculos desenhados pelo programa Ps.

redefinição. Os corpos das funções estão contidos entre os pares sublinhados. Os limites dessa região foram ilustrados na figura 7.54, mas não foram mostrados os ambientes léxicos. Cada um dos quatro ambientes está identificado pelo nome da função — A, B, C ou D — que foi declarada por ele. Cada variável associada a um deles é implicitamente declarada dentro daquele bloco. A parte da saída, o que pode ser obtido respeito de escopo de cada ocorrência de x?

7.5 Escreva um procedimento para inserir um item numa lista ligada passando-se um apontador para a cabeça da lista. Sob que circunstâncias há transmissão de parâmetros esse procedimento faz isso?

7.6 O que é impresso pelo programa na Fig. 7.53, assumindo-se (a) chamada por valor, (b) chamada por referência, (c) ligação de cópia e restauração, (d) chamada por nome?

```
program main(input, output);
    procedure p(x, y, z);
        begin
            x := y + 1;
            y := z - x
        end;
    begin
        a := 2;
        b := 3;
        p(a + b, a, a);
        print a
    end.
```

Fig. 7.53. Para determinar, durante a transmissão de parâmetros.

7.7 Quando um procedimento é transmitido como parâmetro numa ligação em de escopo léxico, seu ambiente não local pode ser mantido junto com o endereço que ele de acesso. Forneça um algoritmo para determinar esse ele.

7.8 Os três tipos de ambientes que podem ser associados a um procedimento passado como parâmetro são ilustrados pelo programa Pascal na Fig. 7.54. Os ambientes [illegible] de um tal procedimento consistem nas ligações de identificadores nos pontos em que o procedimento está definido, transportados como parâmetros ocultos, respectivamente. Considere a função [illegible] como um parâmetro à função [illegible]. Usando os ambientes léxico, de transmissão e de ligação de [illegible] nome mais local [illegible] linha. Sendo, os respectivos declarações de [illegible] as linhas 6, 10 e 5, respectivamente
a) Descubra a árvore de [illegible] para esse programa

```
[illegible]
2 [illegible]
3 [illegible]
4 [illegible]
5 [illegible]
6 [illegible]
7 [illegible]
8 [illegible]
9 [illegible]
10 [illegible]
11 [illegible]
12 [illegible]
13 [illegible]
14 [illegible]
15 [illegible]
```

Fig. 7.54. Exemplos de ambientes léxico, de transmissão e de ligação

b) Qual é a saída do programa, usando-se os ambientes léxico, de transmissão e de ligação para [illegible]?

7.9 Modifique a implementação da função para as regras de escopo léxico de Lisp, a estabelecer o ambiente léxico corretamente quando um procedimento transmitido como parâmetro for ativado.

*7.9 O enfoque usado [illegible] da linha 11 do pseudoprograma da Fig. 7.55 chama a função a, que finalmente a função exibe o valor como resultado

a) Desenhe a árvore de ativações desse programa.

b) Suponha que o escopo léxico seja usado para [illegible] locais. Por que o programa, na falha será a alocação de pilha for usada?

c) Qual é a saída do programa com a alocação de heap?

```
1  program [illegible](output);
2  var f: function [illegible];
3
4      function a: [illegible];
5          var x: integer;
6          function exibe: integer;
7              begin return x end;
8          begin x := 10; return exibe end;
9
10     procedure b(x: function [illegible]);
11         begin writeln(x) end
12
13     begin
14         f := a; b(f)
15     end.
```

Fig. 7.55. Pseudoprograma [illegible] a função [illegible] [illegible]

*7.10 Certas linguagens, como Lisp, [illegible] a habilidade de retornar procedimentos recentemente criados em tempo de execução. Na Fig. 7.56, todas as funções, quer definidas [illegible] pelo programador, quer [illegible] em tempo de execução, recebem um número um argumento e retornam um valor [illegible] função ou um real. O operador [illegible] a composição de funções, isto é, [illegible].

a) Qual é o valor impresso [illegible]?

b) Suponhamos que sempre que um procedimento [illegible] retornado, seu registro de ativação se torna um [illegible] de registro de ativação para a função que retorna p. O ambiente

```
function [illegible]
var p: function;
    function q;
        [illegible] { [illegible] }
        return x
    end { q };

function [illegible];
    return sin
end { h };

function g(x: function);
var p: function;
    [illegible] { [illegible] }
    return p
end { g };

function [illegible];
var x: real;
    [illegible]: function;
    begin
        x := [illegible];
        [illegible] := x;
        x := [illegible];
        print x
    end { [illegible] };
```

Fig. 7.56. Pseudoprograma que cria funções em tempo de execução.

ficiente de transições onde *p* pode ainda ser mantida através de uma árvore de registros de ativação em lugar de uma pilha. Esquematize a árvore de registros de ativação quando é computado por *[illegible]* na Fig. 7.50.

c) Alternativamente, suponha que um registro de ativação para *p* seja criado quando *p* for ativado e que seja tornado filho do registro de ativação para o procedimento chamador *p*. Esse enfoque pode ser usado para manter o ambiente de ativação para *p*. Desenhe instantâneos dos registros de ativação e seus relacionamentos pai-filho na passada em que os comandos em *[illegible]* sejam executados. É uma pilha suficiente para abrigar os registros de ativação quando esse enfoque for usado?

7.11 Outra forma de se tratar a remoção nas tabelas *hash* dos nomes cujo escopo já tenha passado, como na Seção 7.6, é deixar os nomes expirados numa lista, até que a lista seja pesquisada de novo. Assumindo que as entradas incluam o nome do procedimento no qual a declaração é feita, podemos, em princípio, informar se um nome é antigo e removê-lo, se for o caso. Forneça um esquema de indexação para procedimentos que nos habilite informar, num tempo $O(1)$, se um procedimento é "antigo", isto é, se seu escopo já passou.

7.12 Muitas funções *hash* podem ser caracterizadas por uma seqüência de constantes inteiras $\alpha_0, \alpha_1, \ldots$. Se c_i, [illegible], é o valor inteiro do *i*-ésimo caractere na cadeia *s*, então a cadeia é *hashed* para

$$hash(s) = (\alpha_0 + \sum \ldots) \textbf{ mod } m$$

onde *m* é o tamanho da tabela *hash*. Para cada um dos seguintes casos, determine a seqüência de constantes $\alpha_0, \alpha_1, \ldots$ ou mostre que não existe tal seqüência. Cada caso determina um inteiro cujo valor *hash* é obtido tomando-se esse inteiro módulo *m*.

a) Obter a soma dos caracteres.
b) Obter a soma do primeiro e último caracteres.
c) Obter h_i, onde $h_0 = 0$ e $h_i = 2h_{i-1} + c_i$.
d) Tratar os últimos 4 caracteres centrais como um inteiro de 32 bits.
e) Um inteiro de 32 bits pode ser visto como constituído de 4 bytes, onde cada byte é um dígito que toma um dentre 256 possíveis valores. Começando por 0000, para [illegible] adicione [illegible] ao *byte i* **mod** 4, permitindo o transporte de vai-um. Isto é, $c_1, c_5, \ldots$ são adicionados ao *byte* 1, $c_2, c_6, \ldots$ ao *byte* 2 e assim por diante. Retornar o valor final.

*7.13 Por que as funções *hash* caracterizadas por uma seqüência de inteiros $\alpha_0, \alpha_1, \ldots$, como no Exercício 7.12, têm algumas vezes um desempenho pobre se a entrada consistir em cadeias consecutivas como, por exemplo, [illegible]? O sintoma é que, em algum lugar ao longo do caminho, seu comportamento se desvia do randômico e pode ser predito.

**7.14 Quando *n* cadeias podem ser *hashed* para *m* listas, o número médio de cadeias por lista é *n/m*, não importa quão não homogeneamente as cadeias estejam distribuídas. Suponha que *S* seja a distribuição, isto é, uma cadeia randômica é colocada na *i*-ésima lista com probabilidade [illegible]. Suponhamos que uma função de *hash* com distribuição *S* coloque *n* cadeias randomicamente selecionadas na lista *j*, [illegible]. Mostre que o valor esperado $W = \Sigma$ [illegible] $+ 1)/2$ está linearmente relacionado à variância da distribuição *S*. Para uma distribuição uniforme, mostre que o valor esperado de *W* é $(n/2m)(n + 2m - 1)$.

7.15 Suponha que tenhamos a seguinte seqüência de declarações num programa Fortran.

[illegible]

Mostre o conteúdo das áreas de dados para [illegible] mente a parte de [illegible] acessível a partir de [illegible]. Por que não há espaço para [illegible]?

*7.16 Uma estrutura de dados útil para o cômputo de equivalência é a estrutura de *anel*. Usamos um apontador e um campo de deslocamento em cada entrada da tabela de símbolos para ligar membros de um conjunto de equivalência. Esta estrutura é sugerida pela Fig. 7.37, onde *A*, [illegible] e [illegible] são equivalentes e [illegible] também o são, e a localização de [illegible] estando 10 palavras após aquela de [illegible] e assim por diante.

a) Forneça um algoritmo para computar o deslocamento de [illegible] relativo a [illegible], assumindo que [illegible] e [illegible] estão no mesmo conjunto de equivalência.
b) Forneça um algoritmo para computar *inferior* e *superior*, conforme definido na Seção 7.9, relativos à localização de algum nome [illegible].
c) Forneça um algoritmo para processar

[illegible]

Não assuma que [illegible] e [illegible] estejam necessariamente em conjuntos de equivalência diferentes.

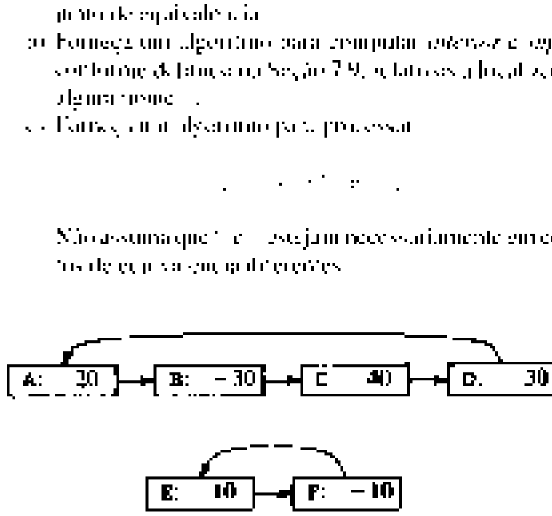


Fig. 7.37. Estrutura em anel.

*7.17 O algoritmo para mapear áreas de dados, fornecido na Seção 7.9, requer que se verifique se o *inferior* [illegible] para *líder* do conjunto de equivalência de [illegible] não faça com que o espaço para o conjunto de equivalência de [illegible] se estenda para antes do começo do bloco [illegible] e que calculemos o valor de *superior* para o líder de [illegible] de forma a incrementar o limite superior do bloco [illegible], se necessário. Forneça fórmulas em termos de *próximo*, o deslocamento de [illegible] no bloco [illegible] e de *último*, a última palavra corrente do bloco, que realizam o teste e atualizem *último*, se necessário.

NOTAS BIBLIOGRÁFICAS

As pilhas desempenharam um papel essencial na implementação de funções recursivas. McCarthy [1981, p. 178] relembra que durante um projeto de implementação de Lisp, começado em 1958, foi decidido "usar uma única pilha contínua e pública para salvar os valores das variáveis e endereços de retorno de subrotinas na implementação das subrotinas recursivas". A inclusão de blocos e de procedimentos recursivos em Algol 60 — ver Naur [1981, Seção 2.10] para uma retrospectiva — dentro de seus projetos — também estimulou o desenvolvimento da alocação de pilha. A ideia de um *display* para se dar acesso aos nomes não locais numa linguagem de escopo léxico é devida a Dijkstra [1960, 1963]. Apesar de Lisp utilizar o escopo dinâmico, é possível se encontrar o efeito do escopo léxico ao se usar de "funarg

dos ponteiros — consistindo em uma função e um elo de acesso. McGowan [1972] descreve o desenvolvimento desse mecanismo. As linguagens sucessoras de Lisp, como Common Lisp (Steele [1984]), têm se afastado do escopo dinâmico.

Explicações a respeito das amarrações para os nomes podem ser encontradas em livros-texto sobre linguagens de programação; ver, por exemplo, Abelson e Sussman [1985], Pratt [1984] ou Tennent [1981]. Uma abordagem alternativa à sugerida no Capítulo 2 é ler a descrição de um compilador. O desenvolvimento passo a passo em Kernighan e Pike [1984] começa com uma calculadora para expressões aritméticas e constrói um interpretador para uma linguagem simples com procedimentos recursivos. Ou ver-se-á o código para Pascal-S em Wirth [1981]. Uma descrição detalhada da alocação de pilha, uso de display e da alocação dinâmica de arrays aparece em Randell e Russell [1964].

Johnson e Ritchie [1981] discutem o projeto da sequência de chamada que permite que o número de argumentos de um procedimento varie de chamada a chamada. Um método geral para se estabelecer um display global é o de seguir a cadeia de elos de acesso, estabelecendo os elementos do display no processo. O enfoque da Seção 7.4 que toca somente um único elemento pode ter sido "bem conhecido" por algum tempo; uma referência publicada é Rohl [1975]. Moses [1970] discute as distinções relevantes entre os ambientes quando uma função é passada como parâmetro e considera os problemas que emergem

[illegible]

quando tais ambientes são implementados usando-se os acessos superficial e profundo. A alocação de pilha não pode ser usada para linguagens com corrotinas ou múltiplos processos. Lampson [1982] considera implementações rápidas usando a alocação de heap.

Na lógica matemática, as variáveis quantificadas de escopo e substituições apareceram com Begriffsschrift de Frege [1879]. A substituição e a transmissão de parâmetros têm sido assuntos de muitos debates, tanto nas comunidades de lógica matemática quanto nas de linguagens de programação. Church [1956, p. 288] observa: "Especialmente difícil é o assunto do enunciado correto da regra da substituição pelas variáveis funcionais", e relata o desenvolvimento de tal regra para o cálculo proposicional. O lambda-cálculo de Church [1941] tem sido aplicado a ambientes de linguagens de programação, por exemplo, por Landin [1964]. Um par consistindo em uma função e um elo de acesso é frequentemente referenciado como um fechamento, seguindo-se a Landin [1964].

Estruturas de dados para as tabelas de símbolos e algoritmos para pesquisá-las são discutidos em detalhes em Knuth [1973b] e Aho, Hopcroft e Ullman [1974, 1983]. A base do hashing é tratada em Knuth [1973b] e Morris [1968b]. O artigo original discutindo o hashing é Peterson [1957]. Mais a respeito da organização de tabelas de símbolos pode ser encontrado em McKeeman [1976]. O Exemplo 7.10 é de Bentley, Cleveland e Sethi [1985]. Reiss [1983] descreve um gerador de tabelas de símbolos.

Os algoritmos de equivalência foram descritos por Arden, Galler e Graham [1961] e Galler e Fischer [1964]; adotamos a última enfoque. A eficiência dos algoritmos de equivalência é discutida em Fischer [1972], Hopcroft e Ullman [1973] e Tarjan [1975].

CAPÍTULO 8

GERAÇÃO DE CÓDIGO INTERMEDIÁRIO

Na tipologia de analisadores sintáticos de um compilador, as madeiras da categoria inferior são o programa-fonte e uma representação intermediária a partir da qual os resultados da retaguarda gerarão o código-alvo. Na medida do possível, os detalhes da linguagem-alvo são confinados aos módulos da retaguarda. Apesar de se poder traduzir o programa-fonte diretamente na linguagem-alvo, algumas das vantagens em se usar uma forma intermediária independente da máquina são:

1. O retargeting automático é facilitado: um compilador para uma máquina-alvo diferente pode ser obtido colocando-se a retaguarda apropriada para a nova máquina.
2. Otimizadores de código independentes da máquina podem ser aplicados à representação intermediária. Tais otimizadores são discutidos e ilustrados no Capítulo 10.

Este capítulo mostra como os métodos dirigidos pela sintaxe dos Capítulos 2 e 5 podem ser usados para traduzir, numa forma intermediária, construções de linguagens de programação, tais como declarações, atribuições e enunciados de fluxo de controle. Por uma questão de simplicidade, assume-se que o programa-fonte já foi analisado sintaticamente e verificado estaticamente, como na organização da Fig. 8.1. As técnicas de definições dirigidas pela sintaxe deste capítulo podem ser implementadas quer durante uma análise sintática *bottom-up*, quer durante uma análise sintática *top-down*, com backtracking. Também são usadas as técnicas do Capítulo 5 de forma que se deseja do a geração de código intermediário pode ser misturada com uma análise sintática.

8.1 LINGUAGENS INTERMEDIÁRIAS

As árvores sintáticas e a notação pós-fixa, introduzidas nas Seções 5.2 e 2.8, representam, respectivamente, duas tipos de representações intermediárias. Uma terceira forma, chamada de código de três endereços, será usada neste capítulo. As árvores sintáticas podem ser úteis na tradução de linguagens de programação, e a notação pós-fixa é usada numa máquina de pilha. Começamos com uma representação mais geral do que qualquer uma delas, que é uma combinação de árvores sintáticas e notação pós-fixa.

Representações Gráficas

Uma árvore sintática descreve a estrutura hierárquica natural de um programa-fonte. Um DDA típico pode associar uma árvore sintática a um programa-fonte, como mostrado na Fig. 8.2.

A notação pós-fixa é uma representação linearizada de uma árvore sintática: uma lista dos nós da árvore na qual um nó aparece imediatamente após seus filhos. A notação pós-fixa para a árvore sintática da Fig. 8.2 é a seguinte:

[illegible] (8.1)

Os nós de uma árvore sintática não aparecem explicitamente na notação pós-fixa. Podem ser recuperados a partir da ordem na qual eles aparecem e do número de operandos que cada operador espera (aridade, respectivamente). A avaliação dos nós de uma expressão na notação pós-fixa é simples: a avaliação ocorre quando um nó aparece, após seus filhos. Ver a Seção 2.8 para mais detalhes e exemplos. Naturalmente, a notação pós-fixa corresponde a uma travessia em ordem pós-fixa de uma árvore sintática.

As árvores sintáticas para os enunciados de atribuição são produzidas pela definição dirigida pela sintaxe da Fig. 5.3, e mostradas esquematicamente na Seção 5.3. O nó interior de uma árvore sintática é rotulado com o nome do operador, e seus filhos representam os operandos. Os dois operadores binários * e + são exemplos de um conjunto completo de operadores numa linguagem típica. As associatividades e precedências dos operadores são implícitas nas árvores sintáticas porque elas foram colocadas na gramática. Uma definição dirigida pela sintaxe constrói a árvore da Fig. 8.2 para a entrada [illegible].

Uma definição dirigida pela sintaxe para produzir o DDA da Fig. 8.2 é mostrada na Seção 8.3. As funções [illegible] criam um nó para um operador binário e um nó para um identificador, respectivamente. Cada função retorna um apontador para um nó recém-criado. Cada nó existente em um DDA é um registro cujos campos representam a informação necessária para interpretar o nó. Em um compilador, o nó também pode conter espaço para o valor ou endereço associado com o nó. Os nós são implementados como registros com um campo para um operador e campos adicionais para ponteiros para os nós filhos. Um operador é também chamado de rótulo do nó. Assim, pode-se referir-se ao operador de um nó como o rótulo do nó. Os nós de uma árvore sintática podem ter campos adicionais para os atributos de uma construção de linguagem de programação. Um nó-folha para um identificador, por exemplo, pode ter um campo adicional que aponta para a entrada da tabela de símbolos relativa ao identificador. Na Seção 8.3, mostramos como uma entrada da tabela de símbolos

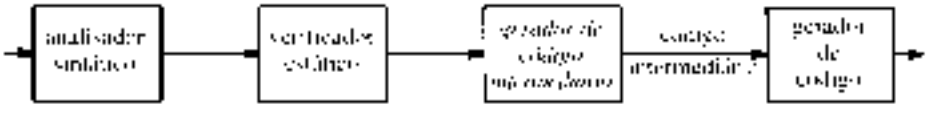


Fig. 8.1 Posição do gerador de código intermediário.

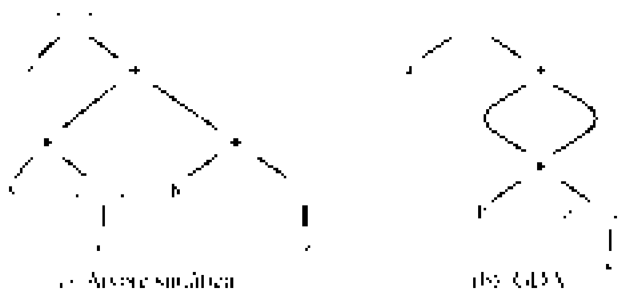


Fig. 8.2. Representações gráficas para a := b * - c + b * - c.

(a) Árvore sintática (b) GDA

Produção	Regras Semânticas
$S \to$ **id** := E	$S.nptr$:= *mknode*('assign', *mkleaf*(**id**, **id**.*place*), $E.nptr$)
$E \to E_1 + E_2$	$E.nptr$:= *mknode*('+', $E_1.nptr$, $E_2.nptr$)
$E \to E_1 * E_2$	$E.nptr$:= *mknode*('*', $E_1.nptr$, $E_2.nptr$)
$E \to - E_1$	$E.nptr$:= *mkunode*('uminus', $E_1.nptr$)
$E \to (E_1)$	$E.nptr$:= $E_1.nptr$
$E \to$ **id**	$E.nptr$:= *mkleaf*(**id**, **id**.*place*)

Fig. 8.3. Definição dirigida pela sintaxe para produzir árvores sintáticas para comandos de atribuição.

pode ser encontrada a partir do atributo **id**.*place*, representando o ponteiro para a entrada da tabela de símbolos de **id**. Se o analisador léxico agrupar todos os lexemas num único array de caracteres, o atributo *nome* poderia ser o índice para o primeiro caractere do lexema.

Duas representações da árvore sintática na Fig. 8.2(a) aparecem na Fig. 8.4. Cada nó é representado como um registro com um campo para seu operador e campos adicionais para os apontadores de seus filhos. Na Fig. 8.4(b), os nós são alocados a partir de um array de registros e o índice ou posição do nó serve como o apontador para o nó. Todos os nós na árvore sintática podem ser visitados seguindo-se os apontadores, começando-se da raiz na posição 10.

Código de Três Endereços

O código de três endereços é uma sequência de enunciados da forma geral

$$x := y \text{ op } z$$

onde x, y e z são nomes, constantes ou objetos de dados temporários gerados pelo compilador; *op* está no lugar de qualquer operador, tal como um operador de aritmética de ponto fixo ou flutuante ou um operador lógico sobre dados booleanos. Note-se que não são permitidas expressões aritméticas construídas, na medida em que só há um operador no lado direito de um enunciado. Por conseguinte, uma expressão da linguagem fonte como $x+y*z$ poderia ser traduzida na sequência

$$t_1 := y * z$$
$$t_2 := x + t_1$$

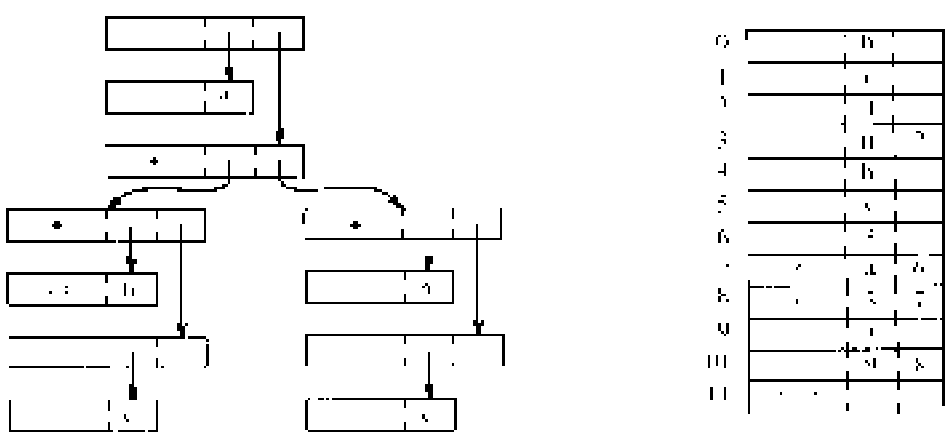
onde t_1 e t_2 são nomes temporários gerados pelo compilador. Esse desmembramento das expressões aritméticas complicadas e de enunciados de fluxo de controle aninhados torna o código de três endereços desejável para a geração do código alvo e a otimização (ver os Capítulos 10 e 12). O uso de nomes para os valores intermediários computados por um programa permite que o código de três endereços seja facilmente rearrumado — diferentemente da notação pós-fixa.

O código de três endereços é uma representação linearizada de uma árvore sintática ou de um GDA, no qual os nomes explícitos correspondem aos nós interiores do grafo. A árvore sintática e o GDA na Fig. 8.2 são representados pelas sequências de código de três endereços na Fig. 8.5. Os nomes de variáveis podem aparecer diretamente nos enunciados de três endereços e, dessa forma, a Fig. 8.5(a) não possui enunciados correspondentes às folhas na Fig. 8.4.

A razão para o termo "código de três endereços" está em que cada enunciado contém usualmente três endereços, dois para os operandos e um para o resultado. Nas implementações do código de três endereços fornecidas mais adiante nesta seção, um nome definido pelo programador é substituído por um apontador para a entrada da tabela de símbolos daquele nome.

Tipos de Enunciados de Três Endereços

Os enunciados de três endereços são semelhantes ao código de montagem. Os enunciados podem ter rótulos simbólicos e existem enunciados para o fluxo de controle. Um rótulo simbólico representa o índice de um enunciado de três endereços num array que abriga o código intermediário. Os índices efetivos podem substituir os rótulos, quer realizando-se uma passagem separada quer utilizando-se a "retrocorreção", discutida na Seção 8.6.



(a) (b)

Fig. 8.4. Duas representações da árvore sintática da Fig. 8.2(a).

(a) Código para a árvore sintática (b) Código para o DAG

Fig. 8.5. Código de três endereços correspondente à árvore e ao DAG da Fig. 8.2

Aqui estão os enunciados de três endereços comuns usados no resto deste livro.

1. Enunciados de atribuição da forma $x = y \; op \; z$, onde op é uma operação binária aritmética ou lógica.

2. Instruções de atribuição da forma $x = op \; y$, onde op é uma operação unária. Dentre as operações unárias essenciais estão os menos unário, a negação lógica, os operadores de deslocamento e os de conversão que, por exemplo, convertem um número de ponto fixo num de ponto flutuante.

3. *Enunciados de cópia* da forma $x = y$, onde o valor de y é atribuído a x.

4. Os desvios incondicional `goto L`. O enunciado de três endereços com rótulo L é o próximo a ser executado.

5. Os desvios condicionais tais como `if x goto L` e `if False x goto L`. Estas instruções são executadas respectivamente quando x é `True` e quando x é `False`, e em seguida executa o enunciado com rótulo L, senão executa a operação seguinte em sequência.

6. `param` x e `call` p, n, para chamadas de procedimentos e `return` y, onde y que representa o valor retornado é opcional. Seu uso típico é na sequência de instruções como a seguinte

```
param x1
param x2
...
param xn
call p, n
```

   gerada como parte de uma chamada de procedimento $p(x_1, x_2, \ldots, x_n)$. O inteiro n, indicando o número de parâmetros atuais em `call` p, n não é redundante porque as chamadas podem ser aninhadas. A implementação de chamadas de procedimentos é discutida na Seção 8.7.

7. Atribuições indexadas da forma $x = y[i]$ e $x[i] = y$. A primeira delas estabelece x como o valor da localização i unidades de memória após a localização de y. O enunciado $x[i] = y$ estabelece o conteúdo da localização i unidades para além de x como o valor de y. Em ambas as instruções, x, y e i se referem a objetos de dados.

8. Atribuições de endereços e de apontadores da forma $x = \&y$, $x = *y$ e $*x = y$. A primeira estabelece o valor de x como sendo a localização de y. Presumivelmente y é um nome, talvez um temporário, que denota uma expressão como $A[i, j]$, tal como x pode ser um apontador ou um temporário. Este é o valor de x é o valor da localização de algum objeto. No enunciado $x = *y$, presumivelmente y é um apontador ou um temporário cujo valor é a localização. O valor de x é igual ao conteúdo daquela localização. Finalmente, $*x = y$ estabelece o valor do objeto apontado por x como o valor de y.

A escolha dos operadores permitidos é uma questão importante no projeto de uma boa forma intermediária. O conjunto de operadores precisa ser suficientemente poderoso para implementar as operações da linguagem fonte. Um pequeno conjunto de operações é mais fácil de implementar numa nova máquina alvo. Entretanto, um conjunto restrito de instruções pode forçar o gerador de código compilador a gerar longas sequências de comandos para algumas operações da linguagem fonte. O otimizador e o gerador de código podem ter que fazer um trabalho árduo se desejarem gerar um código de melhor qualidade.

Tradução Dirigida pela Sintaxe em Código de Três Endereços

Quando o código de três endereços é gerado, os nomes temporários são criados para os nós interiores de uma árvore sintática. Por exemplo, a regra semântica para $E \to E_1 + E_2$ cria o código para computar E numa nova variável temporária. Em geral, o código de três endereços para $id = E$, onde E é geralmente uma operação, consiste em código para avaliar E em alguma variável temporária t seguido pela atribuição $id.lugar = t$. Se uma expressão se constituir em um único identificador, digamos x, então x próprio será usado como a expressão. Para o momento, pelo menos, também criaremos uma nova variável temporária cada vez que uma variável for apontada, mesmo que uma variável temporária não seja necessária. As técnicas para a otimização dos variáveis temporárias são discutidas na Seção 8.5.

A definição S-atribuída na Fig. 8.6 gera código de três endereços para enunciados de atribuição. Dada a entrada $a = b * -c + b * -c$, a definição produz o código da Fig. 8.2(a). O atributo $lugar$ da variável a representa o código de três endereços para a atribuição. O não terminal E possui dois atributos:

1. $E.lugar$, o nome que armazenará o valor de E, e
2. $E.codigo$, a sequência de enunciados de três endereços avaliando E.

A função `novo_temporario` retorna uma sequência de nomes distintos $t_1, t_2, \ldots$, em resposta a pedidos sucessivos efetuados.

Por uma questão de conveniência, usamos a notação geral $E.codigo = E_1.codigo \| E_2.codigo \| \ldots$ na Fig. 8.6 para representar a concatenação de três endereços $E_1, E_2, \ldots$. As expressões ligando em ligar de variáveis, como $E_1.lugar$, são avaliadas quando passadas a funções como `gera` e `lugar` com considerados literalmente. Na prática, os enunciados de três endereços podem ser armazenados para uma saída incremental em vez de serem construídos sobre os atributos $codigo$.

Os enunciados de três endereços podem ser adequadamente gerados pela linguagem de atribuições da Fig. 8.6 através de produções e regras semânticas semelhantes àquelas para as construções $while$ da Fig. 8.7. Na figura, o código para $S \to \mathbf{while}\ E\ \mathbf{do}\ S_1$ é gerado usando novos atributos $S.inicio$ e $S.codigo$ para representar o rótulo do início do código para S e o código para S, respectivamente. Esses atributos representam os rótulos criados pela função `novo_rotulo`, que retorna um novo rótulo a cada vez que for chamada. Note que $S.codigo$ se torna o rótulo do enunciado que vem após o código para a repetição de um laço. Avaliamos E para uma expressão booleana de zero significa verdadeiro, isto é, quando o valor de E se torna zero, o controle deve sair do enunciado $while$.

As expressões que governam o fluxo de controle podem ter regras semânticas e fragmentos de código associados. Os valores relacionais e lógicos podem ser computados como nos enunciados aritméticos, ou podem ser usados para alterar o fluxo de controle através de expressões booleanas.

A tradução pode ser obtida pela adaptação das regras semânticas da Fig. 8.6 (ou veja a Fig. 2.51). A tradução pode ser para um dos

PRODUÇÃO	REGRAS SEMÂNTICAS
$S \rightarrow \mathbf{id} = E$	$S.código = E.código \mid gerar(\mathbf{id}.lugar ':=' E.lugar)$
$E \rightarrow E_1 + E_2$	$E.lugar = novo\_temporário$ $E.código = E_1.código \mid E_2.código \mid$ $gerar(E.lugar ':=' E_1.lugar '+' E_2.lugar)$
$E \rightarrow E_1 * E_2$	$E.lugar = novo\_temporário$ $E.código = E_1.código \mid E_2.código \mid$ $gerar(E.lugar ':=' E_1.lugar '*' E_2.lugar)$
$E \rightarrow -E_1$	$E.lugar = novo\_temporário$ $E.código = E_1.código \mid gerar(E.lugar ':=' \mathbf{uminus}\ E_1.lugar)$
$E \rightarrow (E_1)$	$E.lugar = E_1.lugar$ $E.código = E_1.código$
$E \rightarrow \mathbf{id}$	$E.lugar = \mathbf{id}.lugar$ $E.código = ''$

Fig. 8.6. Definição dirigida pela sintaxe para a tradução de três endereços para atribuições.

identificador propriamente identificado. As regras para as outras produções juntam simplesmente o operador após os códigos para os operandos. Por exemplo, associada à produção $E \rightarrow E_1 * E_2$ está a regra semântica

$$E.código = E_1.código \mid E_2.código \mid gerar(E.lugar ':=' E_1.lugar '*' E_2.lugar)$$

Em geral, a forma intermediária produzida pelos tradutores orientados pela sintaxe deste capítulo podem ser modificadas realizando-se modificações precisas nas regras semânticas.

Implementações de Enunciados de Três Endereços

Enunciados de três endereços são uma representação de código intermediário. Num compilador, esses enunciados podem ser implementados como registros com campos para o operador e os operandos. Três dessas implementações serão apresentadas: as quádruplas, as tuplas e as tuplas indiretas.

Quádruplas

Uma quádrupla é uma estrutura de registro com quatro campos, os quais chamaremos de *op*, *arg1*, *arg2* e *resultado*. O campo *op* contém um código interno para o operador. O enunciado de três endereços $x := y + z$ é representado colocando-se $+$ em *op*, y em *arg1*, z em *arg2* e x em *resultado*. Os enunciados com operadores unários, como $x := -y$ ou $x := y$, não usam *arg2*. Os operadores como **param** não usam nem *arg2* nem *resultado*. Os desvios condicionais e incondicionais colocam o rótulo-alvo em *resultado*. As quádruplas na Fig. 8.8(a) são para a atribuição $a := b * -c + b * -c$. São obtidas a partir do código de três endereços da Fig. 8.5(a).

Os conteúdos dos campos *arg1*, *arg2* e *resultado* são normalmente apontadores para as entradas na tabela de símbolos, dos nomes representados por esses campos. Assim sendo, os nomes podem ser incluídos na tabela de símbolos, à medida que forem encontrados.

Tuplas

Para evitar a substituição de nomes de temporários na tabela de símbolos, poderíamos nos referir a um valor temporário pela posição do enunciado que o computa. Se assim fizermos, os enunciados de três endereços podem ser representados por registros com somente três campos: *op*, *arg1* e *arg2*, como na Fig. 8.8(b). Os campos *arg1* e *arg2*

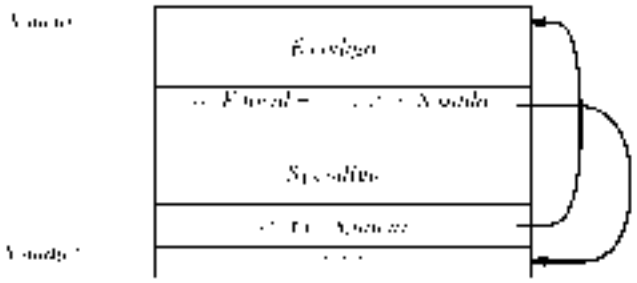
	op	*arg1*	*arg2*	*resultado*
(0)	**uminus**	c		t_1
(1)	*	b	t_1	t_2
(2)	**uminus**	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

(a) Quádruplas

	op	*arg1*	*arg2*
(0)	**uminus**	c	
(1)	*	b	(0)
(2)	**uminus**	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	**assign**	a	(4)

(b) Tuplas

Fig. 8.8. Representações em quádruplas e tuplas de um enunciado de três endereços.



PRODUÇÃO	REGRAS SEMÂNTICAS
$S \rightarrow$ **while** E **do** S_1	$S.início = novo\_rótulo$ $S.falso = novo\_rótulo$ $S.código = gerar(S.início ':') \mid$ $E.código \mid$ $gerar('\mathbf{if}\ ' E.lugar '=' '0'\ '\mathbf{goto}\ ' S.falso) \mid$ $S_1.código \mid$ $gerar('\mathbf{goto}\ ' S.início) \mid$ $gerar(S.falso ':')$

Fig. 8.7. Regras semânticas para gerar código para um laço **while**.

	op	arg 1	arg 2
(0)	[ ]=	x	i
(1)	assign	(0)	y

(a) x[i] := y

	op	arg 1	arg 2
(0)	=[ ]	y	i
(1)	assign	x	(0)

(b) x := y[i]

Fig. 8.9. Novas representações de tuplas

	comando
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

	op	arg 1	arg 2
(0)	uminus	c	
(1)	*	b	(14)
(2)	uminus	c	
(3)	*	b	(16)
(4)	*	(15)	(17)
(5)	assign	a	(18)

Fig. 8.10. Representação de tuplas indiretas de comandos de três endereços

e opcionalmente de argumentos de tipo, são apontadores para a tabela de símbolos para nomes definidos pelo programador ou constantes, ou apontadores para estruturas de tuplas para valores temporários. Uma vez que tenham sido assim usados, esses tuplos deixam de ser interpretados e conhecidos como tuplas. Exceto pelo tratamento dos nomes definidos pelo programador, as tuplas correspondem à representação da árvore sintática em GID. Através de uma passagem de mais, como a Fig. 8.4.

Os números entre parênteses representam apontadores para a entrada de tuplas, enquanto os apontadores para a tabela de símbolos são representados pelos próprios nomes. Na prática, as entradas nos campos op e arg 1 podem ser substituídas por campos que contenham ponteiros como os adotados. As tuplas da Fig. 8.9(b) correspondem às quádruplas da Fig. 8.5(a). Note-se que o comando de cópia x := y e cópia x := y[i] são representados de tuplas exatamente da mesma forma, porque a única diferença é a operação [ ].

Uma operação binária como + [ ] pode ter duas entradas na representação de tuplas, como mostra a Fig. 8.9(b), ou opcionalmente pode ser representada como duas operações, Fig. 8.9(a).

Tuplas Indiretas

Uma complementação do esquema de três endereços que vimos é considerada a que lista apontadores para tuplas, ao invés de listar as próprias tuplas. Essa representação é naturalmente chamada de tuplas indiretas.

Por exemplo, a árvore sintática comum para a lista apontadores para tuplas na ordem desejada. As tuplas então podem ser representadas como na Fig. 8.10.

Comparação das Representações: O Uso da Indireção

A vantagem entre as tuplas e as quádruplas pode ser caracterizada como uma questão sobre a quantidade de indireção que está presente na representação. Quando um otimizador, por exemplo, o código-alvo, cada passo, ele tem a opção de alterar pelo programador, ou atribuída a si alguma localização de memória em tempo de execução. Essa localização seria colocada na tabela de símbolos para o dado. Usando a notação de quádruplas, um comando de três endereços do tipo x := y + z pode ter o seu significado completamente alterado se a localização de uma das variáveis for alterada na tabela de símbolos.

Um benefício mais importante das quádruplas parece ocorrer em compiladores otimizantes, onde as expressões são frequentemente deslocadas de lugar. Usando-se a notação de quádruplas, a tabela de símbolos interpreta o significado de instruções como a computação de um valor específico. Se uma expressão for encontrada que computa o mesmo valor — mesmo operador e mesmos operandos — não há necessidade de computar o valor novamente. Basta-se usar o valor temporário que contém o valor. Isso pode ser feito mantendo-se uma lista de todas as expressões disponíveis e procurando-se nessa lista cada vez que uma nova expressão for encontrada. Se a expressão for encontrada, o valor temporário associado pode ser usado em vez de recomputar a expressão. Esse problema torna-se mais difícil de ser resolvido com tuplas indiretas.

As tuplas indiretas não apresentam tal problema. Um operador pode ser removido pelo simples procedimento de listar a expressão. Como apontadores para as tuplas temporárias são referenciados em outras tuplas, não são mudados. Se algum desses apontadores precisa ser modificado, por conseguinte, as tuplas indiretas se poderão manter com as quádruplas, na qual a alteração de uma variável é propagada. As duas notações representam uma troca de espaço versus velocidade de execução. Como vimos, as tuplas indiretas economizam espaço, a alocação de memória para esses temporários que não necessitam precisar ser posicionada até a fase de geração de código. Entretanto, as tuplas indiretas podem usar algum espaço extra quando comparadas com as quádruplas se a mesma variável temporária for usada mais de uma vez. A razão para isto é que duas entradas na tuplas indiretas, como (14) e (16) da Fig. 8.10 poderiam ser combinadas, como vimos, pois as duas poderiam combinar (15) e (17).

8.2 DECLARAÇÕES

A medida que se aprofunda as declarações num procedimento ou bloco, e vai-se alocando endereços de memória para atribuí-los aos seus locais no procedimento. Para cada nome local, podemos armazenar o tipo e o deslocamento relativo da memória para o nome. O endereço relativo consiste em um deslocamento a partir da base do local estático de dados ou de campo para os dados locais no registro de ativação.

Quando os símbolos de vanguarda do compilador geram código, pode estar com uma máquina alvo em mente. Suponhamos que os endereços de nível consecutivos são alocados a bytes consecutivos na memória, no nível de byte. Os computadores de endereço geram pela vanguarda podem, por conseguinte, incluir multiplicações pelo tamanho de instruções na máquina alvo pode também favorecer certas disposições para os objetos de dados, por conseguinte, de seus endereços. Trataremos aqui o alinhamento de dados, o Exemplo 7.3 mostra como os objetos de dados são alinhados pelos compiladores.

Declarações num Procedimento

Na maioria das linguagens como C, Pascal e Fortran, permite que todas as declarações num único procedimento possam ser processadas como um único grupo. Nesse caso, uma variável global daí em diante pode conhecer o próximo endereço relativo disponível.

Na sequência de tradução da Fig. 8.11 usaremos um símbolo Φ para uma variável de declarações da forma Id . T. Antes da primeira declaração a ser

---

Alguns compiladores tuplas como esta são chamadas de tuplas indiretas, pois a tabela de símbolos é apontada pelo nome e não pelo endereço. Isso permite a alteração de endereços de dados em tempo de execução, mas também pode ser usado para otimização. Veja o exercício 8.3 para mais detalhes.

```
D → ε                          { ts.escopo.numvars := 0 }

    | D

D → D ; D

D → id : T                     { ts.escopo.id.tipo := T.tipo
                                 ts.escopo.id.endereço := ts.escopo.numvars
                                 ts.escopo.numvars := ts.escopo.numvars + T.largura }

T → integer                    { T.tipo := inteiro
                                 T.largura := 4 }

T → real                       { T.tipo := real
                                 T.largura := 8 }

T → array [ num ] of T         { T.tipo := array(num.valor, T.tipo)
                                 T.largura := num.valor × T.largura }

T → ↑ T                        { T.tipo := apontador(T.tipo)
                                 T.largura := 4 }
```

Fig. 8.11 Computação dos tipos e endereços relativos de variáveis locais

contexto do nível anterior é restaurado. À medida que cada nome é reconhecido, ele é inserido na tabela de símbolos com um tipo e um endereço relativo iguais aos valores correntes de t e $offset$. Em uma segunda versão, mostrada pelo fragmento de código de tradução dirigida pela sintaxe a seguir, o tipo e o endereço relativo são inseridos pela ação semântica associada à produção $D \rightarrow id : T$.

```
D → id : T    { top.put(id.lexema, T.tipo, offset);
                offset := offset + T.largura }
```

O procedimento `top.put` cria uma entrada para `id.lexema`, define o tipo dessa entrada como `T.tipo` e o seu endereço relativo como `offset`. Em uma segunda versão, o procedimento `top.put` pode ser chamado para criar uma entrada para o nome `id`. Os atributos sintetizados `tipo` e `largura` são associados aos não-terminais T e X para uso na Seção 6.3.3. Suponha que t_1 e t_2 são os tipos de duas expressões quaisquer. Diferentes operadores de linguagens podem combinar essas expressões de tipos de diferentes maneiras. Uma função `max`(t_1, t_2), por exemplo, pode definir o tipo resultante como o tipo mais geral dentre os dois tipos. Também é comum verificar se t_1 e t_2 são iguais. Uma função `erro` também pode ser definida para sinalizar um erro de tipo. Discutiremos essas regras em detalhes na Seção 6.5.1.

Na Fig. 8.11, os atributos sintetizados $tipo$ e $largura$ de T são usados da seguinte forma. A largura de um vetor é obtida pela multiplicação do tamanho de cada elemento pelo número de elementos do vetor. A largura de um apontador é 4. Em Pascal e C, um apontador pode ser visto como um ponteiro para um objeto apontado, mas também pode ser visto como um índice de uma matriz de objetos, como discutido na Seção 6.1.4. A alocação de memória para tais tipos é mais complexa em linguagens nas quais objetos podem crescer ou encolher durante a vida útil de um programa, ou podem ser criados e destruídos durante a execução.

A implementação de declarações no esquema de tradução da Fig. 8.11 é muito semelhante à de uma atribuição de variável. Com a produção

$$D \rightarrow id : T \quad \{ top.put(id.lexema, T.tipo, T.largura) \} \qquad (8.21)$$

os mesmos terminais que ajustam t, $offset$ e os atributos na Seção 5.6 de maneira semelhante podem ser usados para ajustar as variáveis de forma que todas as ações apareçam nas extremidades direitas das produções. Uma alternativa mais elegante é marcar a produção $M \rightarrow \varepsilon$ (Seção 5.5.4) da seguinte forma:

```
D → M D
M → ε    { salva(ts); ts := novo(ts);
           ts.escopo.numvars := 0 }
```

Controlando o Escopo das Informações

Uma linguagem com procedimentos aninhados, como Pascal, requer que cada procedimento possa ter seus atributos e endereços relativos armazenados em uma tabela de símbolos separada. Quando um procedimento é reconhecido, o processamento de declarações do procedimento envolvente é suspenso temporariamente. Essa abordagem será demonstrada através da adição de regras semânticas à seguinte gramática:

```
P → D
D → D ; D | id : T | proc id ; D ; S
```

As modificações para os não-terminais T para os tipos e tipos não são mostradas porque devem ser iguais às da Fig. 8.11. O não-terminal T possui os atributos sintetizados $tipo$ e $largura$, como no esquema de tradução da Fig. 8.11.

Por facilidade de implementação, supõe-se que existe uma tabela de símbolos separada para cada procedimento na linguagem. Uma implementação simples usa uma pilha de tabelas de símbolos. Uma tabela de símbolos é uma lista ligada de entradas para os nomes. Implementações mais elaboradas podem substituir essa lista linear se isso for desejado.

Uma nova tabela de símbolos é criada quando uma declaração de procedimento $D \rightarrow \mathbf{proc}\ id\ ;\ D_1\ ;\ S$ é reconhecida. Entradas para as declarações em D_1 são criadas na nova tabela. A nova tabela aponta de volta para a tabela de símbolos do procedimento envolvente. Ao ser concluído o processamento de D_1, a nova tabela (contendo as entradas para as declarações em D_1) é armazenada. A tabela apontada de volta é então restaurada.

Uma vez que todos os procedimentos são declarados em algum lugar no programa, a Fig. 8.11 é suficiente para mostrar como as tabelas de símbolos são manipuladas. Suponha que cada procedimento retorne a tabela de símbolos que não mais lhe é necessária. Suponha também que `mktable(previous)` crie uma nova tabela de símbolos cujo apontador de volta seja `previous`, e que `enter(table, name, type, offset)` crie uma nova entrada para `name` na tabela de símbolos apontada por `table`. Também suponha que `addwidth(table, width)` grave a soma da largura de todas as entradas da tabela de símbolos apontada por `table` nos cabeçalhos correspondentes a essa tabela, e que `enterproc(table, name, newtable)` crie uma nova entrada para o procedimento `name` com tabela de símbolos apontada por `newtable`. Novamente, suponha que `table` e `newtable` apontem para objetos de tabela de símbolos.

As regras semânticas são definidas em termos das seguintes operações:

`mktable(previous)` cria uma nova tabela de símbolos e retorna um apontador para a nova tabela. O argumento `previous` aponta para a tabela anteriormente criada, possivelmente nula para o procedimento envolvente. O apontador retornado é colocado em cabeçalho para uma tabela, juntamente com outras informações, tais como a profundidade de aninhamento de um procedimento. Pode-se usar uma contagem numérica para as profundidades de aninhamento, com 1 para declarações globais.

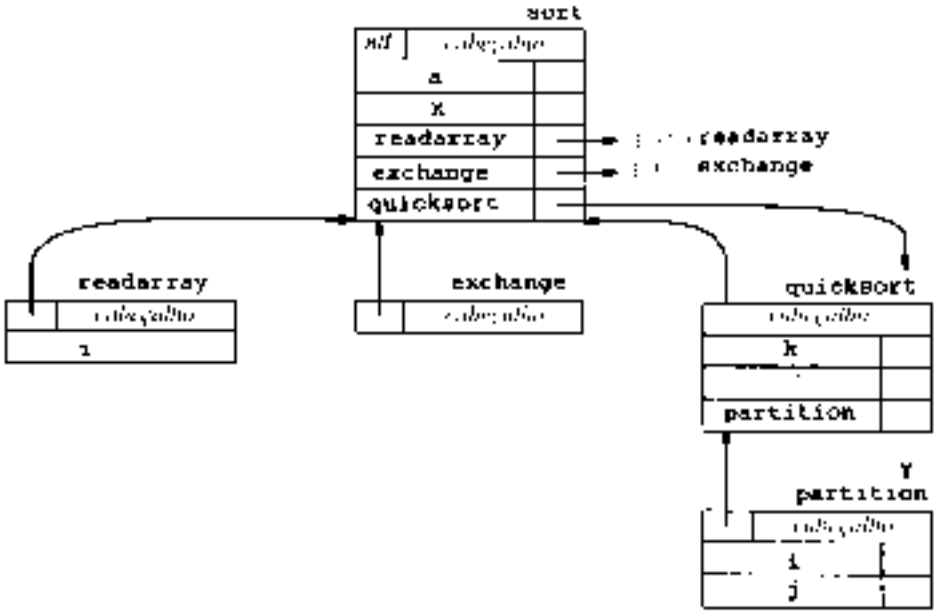


Fig. 8.12. Tabelas de símbolos para procedimentos aninhados

2. *inserir*(*cabeçalho*, *nome*, *tipo*, *deslocamento*) cria uma nova entrada para o nome *nome* na tabela de símbolos apontada por *cabeçalho*. De novo, *nome* é tratado como uma cadeia de caracteres. *tipo* e *deslocamento* são copiados para os campos da entrada criada.

3. *registrar_largura*(*cabeçalho*, *largura*) registra a largura total de todas as entradas de *cabeçalho* no cabeçalho associado a esta tabela de símbolos.

4. *anexar_para*(*cabeçalho1*, *nome*, *cabeçalho2*) cria uma nova entrada para o nome de procedimento *nome* na tabela de símbolos apontada por *cabeçalho1*. O argumento *cabeçalho2* aponta para a tabela de símbolos do procedimento *nome*.

O esquema de tradução na Fig. 8.13 constrói cadeias ligadas pelo campo *cabeçalho* dispostas em uma única passagem, usando uma pilha *per_tab* para guardar os apontadores para as tabelas de símbolos dos procedimentos envolventes. Com as tabelas de símbolos da Fig. 8.12, *per_tab* irá conter apontadores para as tabelas de [illegible] quando as declarações dentro de [illegible] forem consideradas. O apontador para a tabela corrente de símbolos está no topo. A outra pilha, *off*, é uma generalização natural para os procedimentos aninhados com a atributo *deslocamento* (Fig. 8.11). O elemento de topo de *deslocamento* é o próximo endereço relativo disponível no procedimento corrente.

Todas as ações serão feitas nas subárvores para B e C em

$$A \rightarrow B\ C\ \{\text{ação}\}$$

serão realizadas antes que ação seja feita no final da produção, ou seja. Por conseguinte, a ação associada ao marcador M na Fig. 8.13 será a primeira a ser realizada.

A ação para o não-terminal M inicializa a pilha *per_tab* com a tabela de símbolos para o escopo mais externo, criada pela operação *cabeçalho*. A ação também empilha o endereço relativo 0 sobre a pilha *deslocamento*. O não-terminal N desempenha um papel similar quando uma declaração de procedimento aparece. Sua ação cria uma nova tabela de símbolos usando *cabeçalho* para criar uma nova tabela de símbolos. A ação também empilha *cabeçalho* sobre a pilha *per_tab*, de modo que a nova tabela é empilhada acima daquela para o escopo envolvente. De novo, 0 é empilhado sobre a pilha *deslocamento*.

Para cada declaração de variável $\text{id} : T$, uma entrada é criada para id na tabela corrente de símbolos. Esta declaração deixa a pilha *per_tab*

$P \rightarrow M\ D$	{ *registrar_largura*(*topo*(*per_tab*), *topo*(*deslocamento*)); *desempilhar*(*per_tab*); *desempilhar*(*deslocamento*) }
$M \rightarrow \epsilon$	{ t := *criar*(*cabeçalho*); *empilhar*(*t*, *per_tab*); *empilhar*(0, *deslocamento*) }
$D \rightarrow D_1\ ;\ D_2$	
$D \rightarrow \text{proc id}\ ;\ N\ D_1\ ;\ S$	{ t := *topo*(*per_tab*); *registrar_largura*(*t*, *topo*(*deslocamento*)); *desempilhar*(*per_tab*); *desempilhar*(*deslocamento*); *anexar_para*(*topo*(*per_tab*), id.*nome*, *t*) }
$D \rightarrow \text{id} : T$	{ *inserir*(*topo*(*per_tab*), id.*nome*, *T.tipo*, *topo*(*deslocamento*)); *topo*(*deslocamento*) := *topo*(*deslocamento*) + *T.largura* }
$N \rightarrow \epsilon$	{ t := *criar*(); *empilhar*(*t*, *per_tab*); *empilhar*(0, *deslocamento*) }

Fig. 8.13. Processando declarações em procedimentos aninhados

...mbólica corrente pela pilha de símbolos usada anteriormente para os blocos aninhados. Quando aparece uma declaração de D — **proc** id, VD ; S, vemos a lista de todas as declarações geradas por D sendo empilhada na pilha de símbolos. Assim, ao ser feita a busca de um nome local, encontra-se primeiro a declaração mais recente desse nome. As pilhas vão sendo desempilhadas à medida que o compilador termina o processamento de cada procedimento envolvente. Nesse ponto, o nome do procedimento é removido da tabela de símbolos de seu procedimento envolvente.

Nomes de Campos em Registros

A seguinte produção permite que o não-terminal T gere registros ao invés de tipos básicos apontados por `num`:

$$T \rightarrow \textbf{record}\ D\ \textbf{end}$$

As ações no esquema de tradução da Fig. 8.14 criam a tabela de símbolos para os campos dentro de registros, usando uma estrutura de dados como a descrita na Seção 2.3.5. Como as definições de procedimentos não aninhadas em pares de linguagens como a Fig. 8.13, a mesma variável `prev` pode ser usada para as tabelas de símbolos de todos os registros. A função `empilhar` cria uma nova tabela de símbolos, com `prev` apontando para a tabela passada como seu argumento. `empilhar` também cria um cabeçalho para a tabela, onde `prev` é armazenado, e inicializa a contagem de largura para zero. `desempilhar` descarta a tabela de símbolos e guarda a largura de todos os dados dentro do registro após os campos terem sido processados. Veja, no segundo enunciado **end** da Fig. 8.14, como essa largura é armazenada no atributo sintetizado $T.largura$. O tipo $T.tipo$ é simplesmente `registro` aplicado a um apontador para a tabela de símbolos para os campos de registros. Esse apontador será usado na próxima seção para determinar os nomes, tipos e endereços dos campos no registro de $T.tipo$.

8.3 ENUNCIADOS DE ATRIBUIÇÃO

Nesta seção, as expressões podem ser do tipo inteiro, real, ou de registros. Como parte da tradução das atribuições em código de três endereços, mostramos como os nomes podem ser pesquisados na tabela de símbolos e como os elementos de arranjos e registros podem receber ...

Nomes na Tabela de Símbolos

Na Seção 8.1, nomeamos cada dado de três endereços ao nome próprio mesmo, com a simplificação de que esses nomes estavam em lugar dos apontadores para as respectivas entradas na tabela de símbolos. O esquema de tradução na Fig. 8.15 mostra como as entradas da tabela de símbolos podem ser acessadas. O lexema para o nome representado por **id** é fornecido pelo atributo `id.nome`. A operação `procura(id.nome)` verifica se existe uma entrada para esse lexema na tabela de símbolos. Se houver, uma apontador para a entrada para esse nome é retornado; caso contrário, `procura` retorna `nil` para indicar que nenhuma entrada foi encontrada.

As ações semânticas na Fig. 8.15 usam o procedimento `erro` para emitir mensagens de três endereços para algum arquivo de saída ao invés de construir os atributos `código` para os não-terminais, como na Fig. 8.6. A partir da Seção 2.8, a tradução pode ser feita através da emissão para um arquivo de saída se os atributos `código` para os não-terminais nos lados esquerdos das produções forem caracterizados pela concatenação dos atributos `código` dos não-terminais no lado direito, na mesma ordem em que esses não-terminais figuram no lado direito, talvez com algumas cadeias adicionais entremeadas.

Pela nossa interpretação da operação `procura`, o esquema de tradução pode ser usado mesmo que a regra de escopo atribua nomes diferentes a nomes não-locais, como em Pascal. Para uma situação mais concreta, imaginemos que o contexto no qual uma atribuição aparece seja dado pela seguinte gramática.

$$\begin{aligned} P &\rightarrow M\ D \\ M &\rightarrow \epsilon \\ D &\rightarrow D_1\ ;\ D_2 \mid \textbf{id}\ :\ T \mid \textbf{proc id}\ ;\ N\ D_1\ ;\ S \\ N &\rightarrow \epsilon \end{aligned}$$

O não-terminal P se torna o novo símbolo de partida quando essas produções são adicionadas àquelas da Fig. 8.13.

Para cada procedimento gerado por esta gramática, o esquema de tradução na Fig. 8.13 cria uma tabela de símbolos separada. Cada uma dessas tabelas de símbolos possui um cabeçalho contendo um apontador para a tabela de procedimento envolvente (veja a Fig. 8.13 para um exemplo). Quando um enunciado que forma o corpo de um procedimento é examinado, um apontador para a tabela de símbolos para os procedimentos aparece no topo da pilha `ant_pnt`. Esse apontador é empilhado por ações associadas ao não-terminal marcador N ao lado direito de $D \rightarrow$ **proc id** ; $N\ D_1$; S.

Sejam as produções para o não-terminal A aquelas na Fig. 8.15. Os nomes no lado direito gerado por S precisam ter sido declarados no procedimento em que S aparece ou em algum procedimento envolvente. Quando aplicada a cada nome, a função `procura` verifica se existe uma entrada para esse nome na tabela de símbolos no topo da pilha `ant_pnt`. Caso não figure, `procura` usa um apontador no cabeçalho da tabela atual para encontrar a tabela de símbolos para o procedimento envolvente e procura o nome por lá. Se o nome não puder ser encontrado em quaisquer desses escopos, ocorrerá devolução `nil`.

Por exemplo, suponhamos que as tabelas de símbolos sejam como na Fig. 8.12 e que uma atribuição no corpo do procedimento ... esteja sendo examinada. A operação `procura` ... há encontrar uma entrada na tabela de símbolos para ... Uma

$T \rightarrow$ **record** D **end**	{ $t := \textbf{registro}(ant\_pnt.topo)$; $T.largura := largura(ant\_pnt.topo)$; $desempilhar\_pilha(ant\_pnt)$; $desempilhar(ant\_pnt.topo)$ }
$N \rightarrow \epsilon$	{ $t := criar\_tabela(nil)$; $empilhar(t, ant\_pnt)$; $empilhar(0, ant\_pnt)$ }

Fig. 8.14. Cabeçalhos para tabelas de símbolos para nomes de campos em registros.

$S \rightarrow \textbf{id} := E$	{ $p := procura(\textbf{id}.nome)$; **se** $p \neq nil$ **então** $emite(p\ ':='\ E.local)$ **senão** $erro$ }
$E \rightarrow E_1 + E_2$	{ $E.local := novo\ temporário$; $emite(E.local\ ':='\ E_1.local\ '+'\ E_2.local)$; }
$E \rightarrow E_1 [E_2]$	{ $E.local := novo\ temporário$; $emite(E.local\ ':='\ E_1.local\ '['\ E_2.local\ ']')$; }
$E \rightarrow$ ...	{ $E.local := novo\ temporário$; $emite(E.local\ ':='$ ... $E.local)$; }
$E \rightarrow E_1$	{ $E.local := E_1.local$ }
$E \rightarrow$ **id**	{ $p := procura(\textbf{id}.nome)$; **se** $p \neq nil$ **então** $E.local := p$ **senão** $erro$ }

Fig. 8.15. Esquema de tradução para pesquisa de nomes em escopos para atribuições.

vez que [illegible] nas está na tabela de símbolos, procurar [illegible] ir. Usar o apontador [illegible] de símbolos para encontrar a base na tabela de símbolos do procedimento envolvente [illegible].

Reusando os Nomes Temporários

Temos até o seguinte assumido que *novo_temporário* gera um novo nome temporário cada vez que um temporário seja necessário. É útil, especialmente em compiladores otimizantes, se criar realmente um nome distinto a cada vez que *novo_temporário* for chamado. O Capítulo 10 fornece a justificativa para se agir assim. No entanto, os temporários usados para guardar os valores intermediários em computações de expressões tendem a entulhar a tabela de símbolos, e é necessário reservar espaço para abrigar seus valores.

Os temporários podem ser reusados através da modificação de *novo_temporário*. Um enfoque alternativo de se empacotar temporários distintos na mesma localização de memória durante a geração de código é explorado no próximo capítulo.

O grosso dos temporários que denotam dados é gerado durante a tradução dirigida pela sintaxe de expressões, através de regras como aquelas da Fig. 8.15. O código gerado pelas regras para $E \to E_1 + E_2$ possui a forma geral

$$\begin{array}{l}\text{avaliar } E_1 \text{ em } t_1\\ \text{avaliar } E_2 \text{ em } t_2\\ t := t_1 + t_2\end{array}$$

A partir das regras para o atributo sintetizado *E.local* se infere que t_1 e t_2 não são usados em qualquer outro lugar do programa. Os tempos de vida desses temporários estão aninhados como pares de parênteses balanceados. De fato, os tempos de vida de todos os temporários usados na avaliação de E_2 estão contidos nos tempos de vida de t_1. É, por conseguinte, possível modificar *novo_temporário* de forma que use, como se fora uma pilha, um pequeno espaço numa área de dados do procedimento, para abrigar os temporários.

Vamos assumir, simplesmente, que estamos lidando somente com inteiros. Mantemos um contador *c*, inicializado em zero. Sempre que um nome temporário for usado como um operando, decrementar *c* de 1. Sempre que um novo nome de temporário for gerado, usar $\$c$ e incrementar *c* de 1. Note-se que a "pilha" de temporários não é empilhada ou desempilhada em tempo de execução, apesar de armazenamentos e cargas de valores temporários serem programados pelo compilador para acontecer no "topo".

Exemplo 8.1. Consideremos a atribuição

$$x := a * b + c * d - e * f$$

A Fig. 8.16 mostra a sequência de enunciados de três endereços que seria gerada pelas regras semânticas da Fig. 8.15, se *novo_temporário* fosse modificado. A figura também contém uma indicação do valor corrente de *c* após a geração de cada enunciado. Note-se que, quando computamos $\$0 - \$1$, *c* é decrementado para zero e, dessa forma, $\$0$ está disponível, de novo, para abrigar o resultado. □

ENUNCIADO	VALOR DE c
	0
\$0 := a * b	1
\$1 := c * d	2
\$0 := \$0 + \$1	1
\$1 := e * f	2
\$0 := \$0 - \$1	1
x := \$0	0

Fig. 8.16. Código de três endereços com temporários empilhados.

Os temporários que podem ser atribuídos e/ou usados mais de uma vez, por exemplo, numa atribuição condicional, não podem ter os nomes associados numa base último-a-entrar, primeiro-a-sair (LIFO) descrita acima. Como tendem a ser raros, tais valores temporários podem ter os nomes atribuídos de forma particular. O mesmo problema de temporários definidos ou usados mais de uma vez ocorre quando realizamos uma otimização de código, tal como combinar subexpressões comuns ou mover uma computação para fora de um laço (ver o Capítulo 10). Uma estratégia razoável é a de se criar um novo nome sempre que criarmos uma definição adicional ou uso para um temporário ou movermos sua computação.

Endereçando Elementos de Arrays

Os elementos de um *array* podem ser eficientemente acessados se os elementos forem armazenados num bloco de localizações consecutivas. Se a largura de cada elemento do *array* é *w*, o *i*-ésimo elemento do *array* A começa na localização

$$base + (i - low) \times w \tag{8.4}$$

onde *low* é o limite inferior do intervalo de subscritos e *base* é o endereço relativo da memória alocada para o *array*. Isto é, *base* é o endereço relativo de A[*low*].

A expressão (8.4) pode ser parcialmente avaliada em tempo de compilação se for reescrita como

$$i \times w + (base - low \times w)$$

A subexpressão $c = base - low \times w$ pode ser avaliada quando a declaração do *array* for enxergada. Assumimos que *c* é salvo na entrada da tabela de símbolos para A, de forma que o endereço relativo de A[*i*] é obtido simplesmente adicionando-se $i \times w$ a *c*.

A pré-computação em tempo de compilação também pode ser aplicada aos cálculos de endereços de elementos de *arrays* multidimensionais. Um *array* bidimensional é normalmente armazenado em uma de duas formas, ou por linha (linha a linha) ou por coluna (coluna a coluna). A Fig. 8.17 mostra a disposição de um *array* 2 × 3 A (a) organizado por linhas e (b) por colunas. Fortran usa a organização por coluna; Pascal usa a forma por linha, porque A[i, j] é equivalente a A[i][j], e os elementos de cada *array* A[i] são armazenados consecutivamente.

No caso de um *array* bidimensional armazenado na ordem por linha, o endereço relativo de $A[i_1, i_2]$ pode ser calculado pela fórmula

$$base + ((i_1 - low_1) \times n_2 + i_2 - low_2) \times w$$

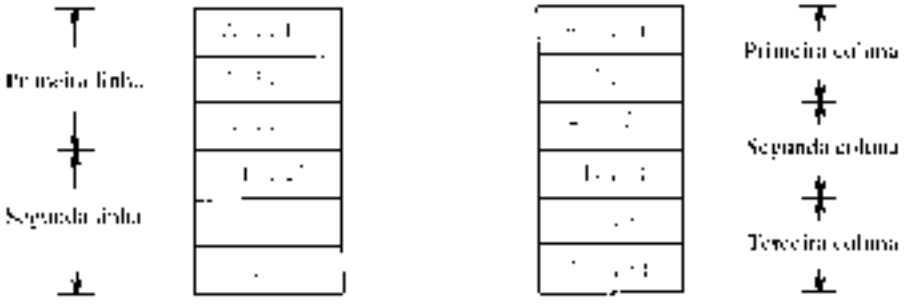
onde low_1 e low_2 são os limites inferiores sobre os valores de i_1 e i_2 e n_2 é o número de valores que i_2 pode assumir. Isto é, se $high_2$ é o limite superior sobre os valores de i_2, então $n_2 = high_2 - low_2 + 1$. Assumindo que i_1 e i_2 sejam os únicos valores que não são conhecidos em tempo de compilação, podemos reescrever a expressão acima como

$$((i_1 \times n_2) + i_2) \times w + (base - ((low_1 \times n_2) + low_2) \times w) \tag{8.5}$$

O último termo desta expressão pode ser determinado em tempo de compilação.

Podemos generalizar a organização por linha ou por coluna para várias dimensões. A generalização da organização por linha significa armazenar os elementos de tal forma que, à medida que esquadrinhamos um bloco de memória, os subscritos mais à direita parecem variar mais rapidamente, como os números num odômetro. A expressão (8.5) se generaliza para a expressão seguinte, para o endereço relativo de $A[i_1, i_2, \ldots, i_k]$

$$((\ldots((i_1 n_2 + i_2) n_3 + i_3) \ldots) n_k + i_k) \times w$$
$$+ \; base - ((\ldots((low_1 n_2 + low_2) n_3 + low_3) \ldots) n_k + low_k) \times w \tag{8.6}$$



(a) Organização por linhas (b) Organização por colunas

Fig. 8.12. Disposições para um array bidimensional.

Daqui para frente, $w = width \times e$ é assumido fixo, e a fórmula na segunda linha de (8.2) pode ser computada pelo compilador a partir de i, j, low_1, low_2 e w, usando apenas uma subtração, uma multiplicação, uma adição e duas deslocações.

Algumas linguagens permitem que os componentes dos arrays sejam parcialmente especificados na declaração, quando o procedimento for chamado em tempo de execução. A declaração de `f` em `int f(int a[][10], int h);` ilustra isso. Em tempo de execução, a função `f` pode ser chamada com até 20 arrays de inteiros como argumentos reais, e a expressão `a[i][j]` é interpretada como `*(a + 10*i + j)`, independentemente do valor de `h`. Como o compilador não pode saber quantos arrays serão passados, ele não pode reservar espaço para eles em tempo de compilação.

Para calcular endereços, a largura de cada array não especificada é omitida da tabela de símbolos, e o endereço de $a[i][j]$ é calculado como $base + i \times w + j \times e$, em tempo de execução. Arrays cujos elementos não ocupam um número contínuo de endereços também podem ser acomodados. Por exemplo, um array pode ser declarado para consistir de números complexos, cada um ocupando duas palavras, mesmo que apenas o número real seja referenciado. Detalhes de tais generalizações são deixados como exercícios.

Representação de Arrays Multidimensionais

Um array com k dimensões pode ser armazenado em um vetor unidimensional. O endereço de um elemento especificado por um subscrito $(i_1, i_2, \ldots, i_k)$ pode ser calculado como

$$base + (i_1 \times n_2 \times \cdots \times n_k) + (i_2 \times n_3 \times \cdots \times n_k) + \cdots + i_k \qquad (8.3)$$

onde $base$ é o endereço relativo do array e n_j é o tamanho da j-ésima dimensão. Os termos da fórmula (8.3) podem ser reagrupados como

$$((\cdots(i_1 n_2 + i_2) n_3 + \cdots) n_k + i_k) + base \qquad (8.4)$$

Para i_1, i_2, ..., i_k, (8.4) permite que o endereço de $a[i_1][i_2]\cdots[i_k]$ seja determinado com k adições e k multiplicações. É provável que muitas expressões envolvendo arrays tenham dimensões pequenas, de modo que a velocidade não seja comprometida muito por um grande número de operações.

Alternativamente, o endereço pode ser calculado com uma adição, uma subtração e k multiplicações, usando a fórmula

$$((\cdots(i_1 n_2 + i_2) n_3 + \cdots) n_k + i_k) - ((\cdots(low_1 n_2 + low_2) n_3 + \cdots) n_k + low_k) + base \qquad (8.5)$$

onde low_j é o limite inferior da subscrita na j-ésima dimensão. Se todos os limites inferiores forem zero, então (8.5) reduz-se a (8.4). Se os limites inferiores forem constantes, mas não zero, então (8.5) envolve uma adição a menos do que (8.4). Mesmo se os limites inferiores não forem constantes, mas o mesmo array for referenciado muitas vezes, como em um laço, algumas das subexpressões de (8.5) podem ser calculadas e armazenadas em variáveis temporárias, de modo que apenas uma adição e uma multiplicação sejam necessárias para cada referência.

Um array pode ter seus elementos armazenados em ordem de linha maior ou coluna maior. Para arrays bidimensionais, a ordem de linha maior é mostrada na Fig. 8.12(a), e a ordem de coluna maior é mostrada na Fig. 8.12(b). Em geral, a ordem de linha maior é a mais natural para linguagens que permitem arrays com limites inferiores diferentes de zero, e a ordem de coluna maior é a mais natural para linguagens que permitem arrays não especificados. Uma razão para isso é que a ordem de coluna maior corresponde à suposição de que os últimos subscritos variam mais rapidamente, como fizemos em (8.2). Essa suposição é importante para arrays não especificados, pois o compilador não tem como determinar quantos elementos existem em uma dimensão não especificada.

Uma razão para preferir a ordem de linha maior é que é mais fácil determinar os endereços de elementos quando alguns dos subscritos mais à esquerda são fixados. Também por essa razão, a ordem de linha maior é preferida em outras linguagens. Por exemplo, se i_1 for fixado em (8.4), o valor de

$$(i_1 n_2) + base$$

pode ser calculado uma vez e reutilizado para diferentes valores de i_2. O valor de (8.4) pode ser recalculado a partir do valor anterior com uma adição e uma multiplicação, pois ele é equivalente a

$$((\cdots(i_1 n_2 + i_2) n_3 + \cdots) n_k + i_k) + base$$

Uma fórmula semelhante pode ser usada para a ordem de coluna maior.

O Esquema de Tradução para o Endereçamento de Elementos de Array

As ações semânticas serão incluídas na gramática:

(1)	S	$\rightarrow$	$L = E$
(2)	E	$\rightarrow$	$E + E$
(3)	E	$\rightarrow$	(E)
(4)	E	$\rightarrow$	L
(5)	L	$\rightarrow$	$Elist\ ]$
(6)	L	$\rightarrow$	**id**
(7)	$Elist$	$\rightarrow$	$Elist, E$
(8)	$Elist$	$\rightarrow$	$\mathbf{id} \mid E$

Como no caso das expressões sem referências a arrays, o código de três endereços em si é produzido pelo procedimento *emit*, não visado pelas ações semânticas.

Cada não-terminal tem um atributo *location*. Quando L é um nome simples, a informação relevante na localização de entrada por L em caso contrário é

```
if L.offset = null then /* L é um nome simples */
    L.location := identificador simples de L
else
    L.location := gen(L.array '[' L.offset ']')
```

O código para as expressões aritméticas é exatamente o mesmo que na Fig. 8.15.

```
(3)  E := E1 + E2    { E.location := novo temporário;
                       emit(E.location '=' E1.location '+' E2.location) }
```

Um compilador pode ter que produzir código para declarações como `float x[100][100], y[10][20];`. Nesse caso, ele deve reservar espaço para `x` e `y` em tempo de compilação. Arrays podem ser declarados para ter tipos de elementos mais complexos, como `struct {int tag; float value;} a[100];`, em que o tipo de elemento é um registro. Nesse caso, o compilador também deve reservar espaço em tempo de compilação, exceto para arrays não especificados.

Um array não especificado é um array cujo tamanho é omitido em uma declaração. Arrays não especificados podem ser passados para procedimentos. Em C, todos os arrays são passados por referência, e os tamanhos dos arrays devem ser especificados posteriormente em uma declaração ou como argumentos para o procedimento.

```
A.sdd
```

Quando uma referência a um nome x é resolvida a L, queremos o valor de L. Portanto, usamos a notação para obter o conteúdo da localização $L.local$ do armazenamento.

```
L.sdd
```

Abaixo, $L.desalocamento$ é um inteiro representando a primeira metade de 8 bytes alocados para um registro de ativação. $L.local$ é um endereço, também com 8 bytes, $L.local$ representa o segundo termo de 16 bytes, retornado pela função $criaNovoLocal()$.

```
L.sdd
```

Um desalocamento muito parecido com um simples.

```
L.sdd
```

Quando aparecer, a expressão de índice for encapsulada aplicando a fórmula de recorrência (8.8). Na ação seguinte, $base.local$ corresponde a x e $c \times W$ é $novo\_local$ a c. Note-se que se $base.local$ possuir mais componentes, então $base.L$ não deve ser usado na produção, pois não os computamos.

```
L.sdd
```

$base.local$ carrega tanto o valor da expressão E quanto o valor de $c \times W$ para $a = 1$.

```
L.sdd
```

Exemplo 8.2. Seja a um array $[0..2] \times [0..3]$ com $tam_1 = tam_2 = 4$. Para calcular $a_{i_1}[i_2]$ precisamos calcular $i_1 \times 4 + i_2$. Um árvore gramatical anotada para a atribuição $x = a[y][z]$ é mostrada na Fig. 8.18. A atribuição é traduzida na seguinte sequência de instruções de três endereços:

```
sdd
```

Para cada variável usamos seu nome em lugar de $id.local$. ■

Conversões de Tipo dentro de Atribuições

Na gramática deste capítulo existem dois tipos diferentes de variáveis escalares, de forma que o compilador possa gerar instruções de três endereços com operadores apropriados.

Para simplificarmos a gramática para os comandos de atribuição como acima, não especificamos que existam dois tipos — real e inteiro — com os inteiros convertidos para reais quando necessário.

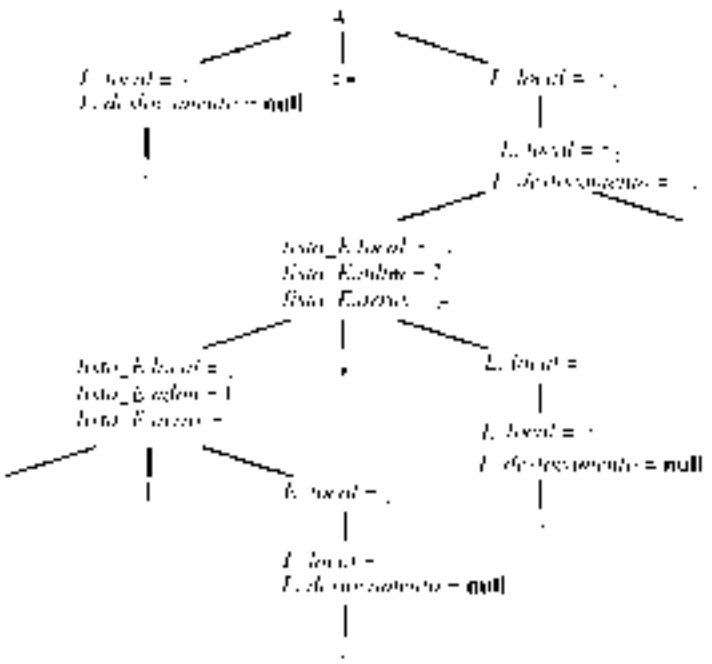


Fig. 8.18. Árvore gramatical anotada para $x = a[y][z]$.

Introduzimos o atributo *E.tipo*, cujo valor é o tipo de *E*. A regra semântica para *E.tipo* associada à produção $E \to E_1 + E_2$ é

```
E → E1 + E2     {E.tipo :=
                    se E1.tipo = inteiro e
                       E2.tipo = inteiro então inteiro
                    senão real}
```

Estratégias correspondentes às Seções 6.1 e 6.2 podem ser aqui adotadas. Omitimos as verificações para os casos de tipo, e a discussão da conversão de tipos aparece no Capítulo 6.

Toda a regra semântica para $E \to E_1 + E_2$ e para outras produções aritméticas pode ser modificada para gerar, quando necessário, endereços de instruções da forma `t := t1 + t2`, em que `t1` e `t2` são endereços de *E*1 e *E*2, respectivamente, e `t` é um endereço para um novo valor calculado. Discutimos também instruções desse tipo no código de operador, na aplicação de memória protegida, se de ponto fixo ou flutuante. A ação semântica completa para uma produção da forma $E \to E_1 + E_2$ é estudada na Fig. 8.19.

Por exemplo, para a entrada

```
x + y * z
```

usando que `x` e `y` sejam tipo *real* e que `z` seja tipo *inteiro*, a saída se parecerá com o seguinte

```
t1 := y * z
t2 := x + t1
```

A ação semântica da Fig. 8.19 passa dois tipos: *inteiro* e *real*. Acarreta também *E*. À medida que o número de tipos sujeitos à conversão aumenta, o número de casos que se tornam necessários também aumenta. Em vez de escrever operadores com mais de dois argumentos. Por conseguinte, com um número amplo de tipos, uma organização diferente das ações semânticas se torna importante.

O Acesso aos Campos nos Registros

Examinamos previamente os tipos que são endereços de memória dos campos de um registro. Uma vantagem de se armazenar essas informações é o acesso da tabela de símbolos para nomes dos campos existir em qualquer lugar, para procurar por campos na tabela de símbolos também pode ser usada para nomes de campos. Como alternativa, uma tabela de símbolos separada pode ser criada para cada tipo de registro, pelas ações semânticas na Fig. 8.14 na última seção. Será de interesse saber para a tabela de símbolos para um tipo de registro contém o tipo correto, tratado pelas ações que constroem registros ao operador e retornado como *E.tipo*.

Examinamos a expressão

```
x.info + 1
```

para discutir como um operador para a tabela de símbolos pode ser usado para o atributo *E.tipo*. A partir das operações nesta expressão segue que `x` precisa ser um apontador para um registro com um nome de campo `info`. Esse tipo é armazenado na tabela de símbolos como `x`. Na Figs. 8.13 e 8.14, o tipo de `x` precisa ser buscado para uma expressão de tipo

```
apontador de registro(t)
```

O tipo de `t` é por conseguinte *registro*, a partir do qual pode ser extraído. O nome de campo `info` é procurado na tabela de símbolos apontada por `t`.

8.4 EXPRESSÕES BOOLEANAS

Nas linguagens de programação, as expressões booleanas têm dois propósitos primários. São usadas para computar valores lógicos, mas mais frequentemente são usadas em expressões condicionais em enunciados que alteram o fluxo de controle, tais como *if-then-else* ou *while-do*.

As expressões booleanas são compostas por operadores booleanos **and**, **or** e **not** aplicados a elementos que são variáveis booleanas ou expressões relacionais. Por sua vez, as expressões relacionais são da forma E_1 relop E_2, onde E_1 e E_2 são expressões aritméticas. Algumas linguagens, tais como PL/I, permitem que expressões mais gerais, onde os operadores booleanos, aritméticos e relacionais podem ser aplicados a expressões de qualquer tipo, sendo distinção entre valores booleanos e outros pelo contexto em que a expressão é encontrada ou necessária. Nesta seção, consideramos as expressões booleanas geradas pela seguinte gramática

$E \to E$ **or** $E \mid E$ **and** $E \mid$ **not** $E \mid (E) \mid$ **id relop id** | **true** | **false**

Usamos **or** e **and** para os operadores lógicos ∨ e ∧, respectivamente. Uma expressão ou subexpressão delimitada por **and** ou **or** pode ser avaliada apenas o suficiente para determinar se a expressão é verdadeira ou falsa. Por exemplo, na expressão `a or b`, `b` não precisa ser avaliado se `a` for verdadeiro. Em `a and b`, `b` não precisa ser avaliado se `a` for falso. Operadores relacionais são analisados apenas para determinar seu valor. Por exemplo, em `a < b or c < d and e < f`, a expressão é equivalente a `(a < b) or ((c < d) and (e < f))`, pela precedência usual dos operadores **and** e **or** e pela regra de que **and** tem precedência sobre **or**.

Métodos de Tradução de Expressões Booleanas

Existem dois métodos principais para se representar o valor de uma expressão booleana. O primeiro método codifica verdadeiro e falso numericamente e avalia uma expressão booleana de forma análoga a uma expressão aritmética. Frequentemente, 1 é usado para denotar verdadeiro e 0 falso, apesar de muitas outras codificações serem também possíveis. Por exemplo, poderíamos fazer qualquer quantidade diferente de zero denotar verdadeiro e zero denotar falso, ou poderíamos fazer qualquer quantidade não negativa denotar verdadeiro e qualquer quantidade negativa, falso.

Nesta seção, examinamos somente uma parte mínima das expressões booleanas que podem ser encontradas em linguagens de programação. Ao final do capítulo, discutimos outras direções que podem ser tomadas. Veja os exercícios para mais informações.

O segundo método principal de se implementar expressões booleanas é através do fluxo de controle, isto é, representar o valor de uma expressão booleana através de uma posição atingida num programa. Esse método é particularmente conveniente na implementação de expressões booleanas em enunciados de fluxo de controle, tais como *if-then-else* e *while-do*. Por exemplo, dada a expressão E_1 or E_2, se determinarmos que E_1 é verdadeira poderíamos concluir que a expressão inteira também é, sem ter que avaliar E_2.

A semântica da linguagem de programação determina se todas as partes de uma expressão booleana precisam ser avaliadas. Se a definição da linguagem permite (ou requer) que partes de uma expressão booleana sejam inavaliadas, o compilador pode otimizar a avaliação das expressões booleanas computando somente o suficiente de uma expressão para determinar o seu valor. Por conseqüência, uma expressão como E_1 **or** E_2 (quer E_1 quer E_2 precisam ser avaliadas por interpretes, mantendo se E_1 for verdadeira) pode ser completada sem avaliar E_2. Se E_2, por exemplo, contenha uma função que modifique uma variável global, uma resposta inesperada pode ser obtida.

Na maioria dos métodos usados é unicamente superficial ao compilador. Por exemplo, o compilador-compilador **BLISS** [Wulf et al. 1975] deixa ao usuário escolher o método apropriado para cada expressão individualmente. Inicialmente, consideraremos ambos os métodos para a tradução de expressões booleanas para o código de três endereços.

Representação Numérica

Vamos primeiro considerar a implementação de expressões booleanas usando 1 para denotar verdadeiro e 0 para falso. As expressões serão avaliadas completamente, da esquerda para a direita, numa forma semelhante a expressões aritméticas. Por exemplo, a tradução para

```
a < b or c < d and e < f
```

é a sequência de três endereços

```
t1 = a < b
t2 = c < d
t3 = e < f
t4 = t2 and t3
t5 = t1 or t4
```

Uma expressão relacional tal como `a < b` é equivalente ao enunciado condicional `if a < b then 1 else 0`, que pode ser traduzido, segundo o esquema de três endereços, como a seqüência

```
100: if a < b goto 103
101: t = 0
102: goto 104
103: t = 1
104:
```

Um esquema de tradução para produzir código de três endereços para expressões booleanas é mostrado na Fig. 8.21. Nesse esquema, assumimos que *emite* coloca enunciados de três endereços num arquivo de saída no formato correto, que *próximo_gerado* fornece o índice do próximo enunciado de três endereços na seqüência de saída e que *cria_rótulo* cria um rótulo após produzir cada enunciado de três endereços.

Exemplo 8.3. O esquema na Fig. 8.20 gerará o código de três endereços da Fig. 8.21 para a expressão `a < b or c < d and e < f`.

Código em Curto-Circuito

Podemos também traduzir uma expressão booleana em código de três endereços sem gerar código para qualquer dos operadores booleanos

$E \rightarrow E_1$ **or** E_2	{ $E.local$ = *novo_temporário*; *emite*($E.local$ ':=' $E_1.local$ 'or' $E_2.local$); }
$E \rightarrow E_1$ **and** E_2	{ $E.local$ = *novo_temporário*; *emite*($E.local$ ':=' $E_1.local$ 'and' $E_2.local$); }
$E \rightarrow$ **not** E_1	{ $E.local$ = *novo_temporário*; *emite*($E.local$ ':=' 'not' $E_1.local$); }
$E \rightarrow (E_1)$	{ $E.local$ = $E_1.local$; }
$E \rightarrow$ **id** **relop** **id**	{ $E.local$ = *novo_temporário*; *emite*('if' **id**$_1$.*local* **relop.op** **id**$_2$.*local* 'goto' *próximo_gerado* + 3); *emite*($E.local$ ':=' '0'); *emite*('goto' *próximo_gerado* + 2); *emite*($E.local$ ':=' '1'); }
$E \rightarrow$ **true**	{ $E.local$ = *novo_temporário*; *emite*($E.local$ ':=' '1'); }
$E \rightarrow$ **false**	{ $E.local$ = *novo_temporário*; *emite*($E.local$ ':=' '0'); }

Fig. 8.20. Esquema de tradução para uma expressão booleana baseada em valores numéricos.

e sem ter código para avaliar a expressão por inteiro. Isso está de avaliação é algumas vezes chamada de código em "curto circuito" ou código de "junção-nula". É possível se avaliar expressões booleanas sem gerar código para os operadores booleanos **and**, **or** e **not** se nós mantivermos o valor de uma expressão por meio pesquisando o seu valor código. Por exemplo, na Fig. 8.21, poderemos dizer que o valor `a < b` é verdadeiro se o enunciado `100: if a < b goto 103` atingir o valor de `t` é verdadeiro. Para muitas expressões booleanas, é possível determinar o valor da expressão sem termos que avaliar todos os componentes.

Enunciados de Fluxo de Controle

Consideremos agora a tradução de expressões booleanas em código de três endereços no contexto de enunciados `if-then`, `if-then-else` e `while-do` tais como aqueles gerados pela seguinte gramática:

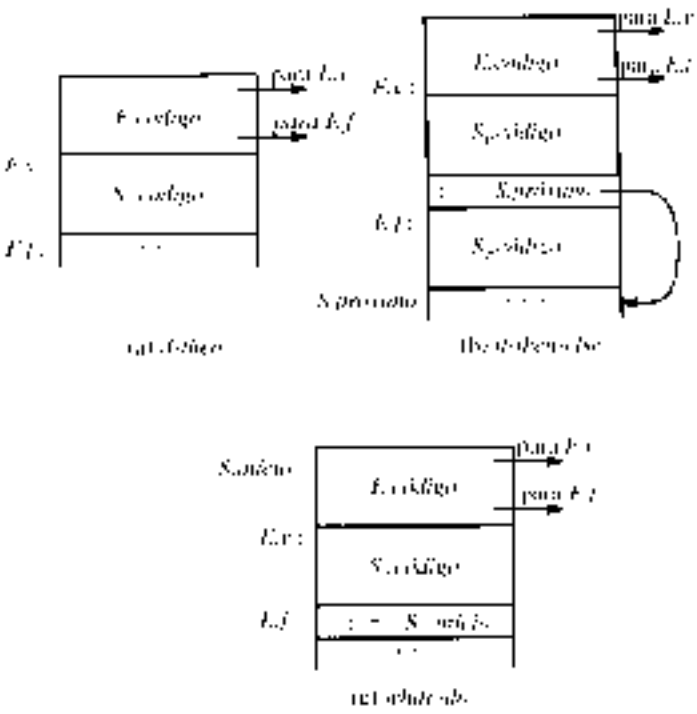
```
S → if E then S1
    if E then S1 else S2
    while E do S1
```

Em cada uma dessas produções, E é uma expressão booleana a ser traduzida. Na tradução, assumimos que um enunciado de três endereços possa ser simbolicamente rotulado e que a função *novo_rótulo* retorna um novo rótulo simbólico cada vez que for chamada.

A uma expressão booleana E associamos dois rótulos, $E.true$ o rótulo para o qual fluir se E for verdadeiro e $E.false$ o rótulo para o qual fluir se E for falso. As regras semânticas para traduzir um enunciado de fluxo de controle S permitem que o controle flua da tradução S código para a instrução de três endereços que se segue imediatamente

```
001                                    103:
01                                     108:
02:                                    109:
05:                                    110:
04:                                    111:
05:                                    112:
06:                                     13:
```

Fig. 8.21. Tradução de



(a) Código (b) Declaração

Fig. 8.22 Código para enunciados semelhantes a enunciados

simplesmente atribuímos as saídas falsas de E_1 para o início das verdadeira e falsa de E. Uma definição dirigida pela sintaxe que gera código de três endereços para expressões booleanas desta forma é mostrada na Fig. 8.24. Note-se que os atributos *true* e *false* são herdados.

Exemplo 8.4. Vamos considerar de novo a expressão

Supondo-se que as saídas verdadeira e falsa para toda a expressão tenham sido estabelecidas em [illegible] e [illegible], a tradução dirigida pela definição da Fig. 8.24, obtém-se o seguinte código:

Note-se que o código gerado não é ótimo, na medida em que o segundo enunciado pode ser eliminado sem mudar seu valor. Instruções redundantes desta forma podem ser subseqüentemente removidas por um simples tratamento *peephole* (ver Capítulo 9). Um método evitar que se crie a geração dessas instruções redundantes é de tratar uma expressão relacional da forma **id**$_1$ **relop id**$_2$ no enunciado [illegible] **id**$_1$ **relop id**$_2$ **goto** [illegible] como a abreviação de [illegible] **relop id**$_2$ **goto** [illegible] **goto** [illegible], com a pressuposição de que, quando a relação for verdadeira, seguimos em frente ao código.

Exemplo 8.5. Consideremos o enunciado

PRODUÇÃO	REGRAS SEMÂNTICAS
$S \rightarrow$ **if** E **then** S_1	$E.true := newlabel()$ $E.false := S.next$ $S_1.next := S.next$ $S.code := E.code \mid\mid gen('if' E.addr 'goto' E.true) \mid\mid gen('goto' S.next) \mid\mid S_1.code$
$S \rightarrow$ **if** E **then** S_1 **else** S_2	$E.true := newlabel()$ $E.false := newlabel()$ $S_1.next := S.next$ $S_2.next := S.next$ $S.code := E.code \mid\mid gen('if' E.addr 'goto' E.true) \mid\mid S_1.code \mid\mid gen('goto' S.next) \mid\mid gen(E.false ':') \mid\mid S_2.code$
$S \rightarrow$ **while** E **do** S_1	$begin := newlabel()$ $E.true := newlabel()$ $E.false := S.next$ $S_1.next := begin$ $S.code := gen(begin ':') \mid\mid E.code \mid\mid gen(E.true ':') \mid\mid S_1.code \mid\mid gen('goto' begin)$

Fig. 8.23. Definição orientada pela sintaxe para comandos de fluxo de controle.

A definição orientada pela sintaxe acima é explorada nos esquemas para os exemplos de atribuição e de expressões booleanas mostrados na seção seguinte. O código

[illegible]

Notamos que os rótulos previstos nessas regras podem ser eliminados pela modificação apropriada dos testes. Essa transformação local tipicamente pode ser feita pela otimização *peephole* discutida no Capítulo 9.

Expressões Booleanas em Modo Misto

É interessante comparar métodos para os simplificados gramática para as expressões booleanas. Na prática, as expressões booleanas frequentemente contêm subexpressões aritméticas, como em [illegible]. Em linguagens onde falso possui o valor numérico de zero e verdadeiro o valor 1, a expressão [illegible] pode ser vista como sendo considerada aritmética, com valor 0 se [illegible] for falso e 1 em caso contrário.

PRODUÇÃO	REGRAS SEMÂNTICAS
$E \rightarrow E_1$ **or** E_2	$E_1.true := E.true$ $E_1.false := newlabel()$ $E_2.true := E.true$ $E_2.false := E.false$ $E.code := E_1.code \mid\mid gen(E_1.false ':') \mid\mid E_2.code$
$E \rightarrow E_1$ **and** E_2	$E_1.true := newlabel()$ $E_1.false := E.false$ $E_2.true := E.true$ $E_2.false := E.false$ $E.code := E_1.code \mid\mid gen(E_1.true ':') \mid\mid E_2.code$
$E \rightarrow$ **not** E_1	$E_1.true := E.false$ $E_1.false := E.true$ $E.code := E_1.code$
$E \rightarrow (E_1)$	$E_1.true := E.true$ $E_1.false := E.false$ $E.code := E_1.code$
$E \rightarrow$ **id**$_1$ **relop id**$_2$	$E.code := gen('if' id_1.addr relop.op id_2.addr 'goto' E.true) \mid\mid gen('goto' E.false)$
$E \rightarrow$ **true**	$E.code := gen('goto' E.true)$
$E \rightarrow$ **false**	$E.code := gen('goto' E.false)$

Fig. 8.24. Definição dirigida pela sintaxe para produzir código de três endereços para expressões booleanas.

Um modo para se representar expressões booleanas sabiamente otimizadas é ainda ser descrito. Vamos supor que as expressões aritméticas também sejam avaliadas por código para computar seus valores. Por exemplo, as variáveis E_1 e E_2 em uma produção

$$E \rightarrow E_1 + E_2 \quad E.\mathbf{and}\ E \rightarrow E.\mathbf{and}\ E \ \ E.\mathbf{relop}\ E \mid \mathbf{id}$$

Podemos supor que $E \rightarrow E_1 + E_2$ produz como resultado aritmético uma soma, mas também operações de outros tipos aritméticos podem ser consideradas, mais complicadas, tais como concatenação de cadeias, onde os valores podem ser descritores de cadeias. Os operandos das expressões E **and** E e E **relop** E produzem valores booleanos representados pelos dois valores de controle. A expressão E **and** E supõe que os dois argumentos sejam booleanos, mas a operação E **relop** E recebe valores dos dois tipos de argumentos, incluindo valores aritméticos. $E \rightarrow$ **id** é também considerada aritmética, apesar de que pode-mos estender esse exemplo para incluir identificadores booleanos.

Para gerar o código nessa situação, podemos usar o atributo sintetizado $E.type$, que pode ser $arith$ ou $bool$, dependendo do tipo de E. Precisamos também dos atributos herdados $E.true$ e $E.false$ para as expressões booleanas e também sintetizado $E.local$ para as expressões aritméticas. Para a regra semântica para $E \rightarrow E_1 + E_2$ é mostrada na Fig. 8.25.

No caso de modo misto, geramos o código para E_1, em seguida para E_2, seguido pelos três enunciados

```
E1.local = 'temp' || newtemp();
gen(E1.code || E2.code || E.local '=' E1.local '+' E2.local);
```

O primeiro enunciado computa o valor E_1 para E_1 quando E_1 for booleano, ou recebe o valor E_1 quando E_1 for falso. O segundo enunciado faz um teste sobre o terceiro. As regras semânticas nos casos restantes são semelhantes às outras produções com somas e as deixamos como exercícios.

```
E.type = arith;
se E2.type = arith e E3.type = arith então início
    /* operação aritmética de adição normal */
    E.local = novo temporário;
    E.código = E2.código || E3.código ||
        gera(E.local '=' E2.local '+' E3.local)
fim
senão se E2.type = arith e E3.type = bool então início
    E2.local = novo temporário;
    E2.true = novo número;
    E2.false = novo número;
    E.código = E2.código || gera('if' E2.local '=' '1' 'goto' E2.true) ||
        gera('goto' E2.false) || gera(E2.true ':' E.local '=' '1') ||
        gera('goto' E3.true) || gera(E2.false ':' E.local '=' '0')
senão se ...
```

Fig. 8.25. Regra semântica para a produção $E \rightarrow E + E$.

8.5 ENUNCIADOS DE DESVIO MÚLTIPLO*

Terminaremos o capítulo com uma discussão de uma variedade de linguagens de programação de desvio múltiplo. Os enunciados **case** ou **switch** podem ser considerados como variedades do enunciado de desvio múltiplo. Uma sintaxe de enunciado de desvio múltiplo é mostrada na Fig. 8.26.

Existe uma expressão seletora, que deve ser avaliada, segundo os valores constantes que a expressão pode assumir, possivelmente incluindo um valor "default", que sempre se iguala ao valor da expressão se nenhum outro valor fizer. A instrução desejada para um desses múltiplos fluxos de código pode ser encontrada como segue:

1. Avaliar a expressão.
2. Encontrar que valor na lista de alternativas é igual ao valor da expressão. Relembremos que o valor *default* sempre se iguala ao da expressão se nenhum dos valores explicitamente mencionados nas alternativas se igualar.
3. Desviar para o código associado ao valor encontrado.

O passo 2 é a parte mais difícil do código que pode ser implementada em muitas diferentes formas, nas possíveis. Se o número de alternativas não for muito grande, digamos no máximo 10, e o compilador usar uma sequência de desvios condicionais, cada um dos quais testa por um valor individual e transfere o controle para o código do enunciado correspondente.

Uma forma mais compacta de implementar essa sequência de desvios condicionais é criar uma tabela de pares, cada qual consistindo em um valor e um rótulo para o código do enunciado como primeiro. O código é gerado de forma a colocar, ao final dessa tabela, o valor da própria expressão seletora mais o rótulo para o enunciado *default*. Um laço simples pode ser gerado pelo compilador de modo a comparar o valor da expressão com cada valor da tabela, estando assegurado que, se nenhuma igualdade for encontrada, a última entrada (*default*) é garantida casar com o valor da expressão.

Se o número de valores exceder a 10, ou algo em torno, é mais eficiente construir uma tabela hash (ver Seção 7.6) para os valores, com o rótulo dos vários enunciados como entradas. Se nenhuma entrada para o valor computado para a expressão do enunciado de desvio múltiplo for encontrada, um salto para o enunciado *default* pode ser gerado.

Existe um caso especial comum para o qual existe uma implementação ainda mais eficiente do desvio de computação múltipla. Se todos os valores possíveis são algum pequeno intervalo de inteiros, digamos de i_{min} até i_{max}, e o número de valores for uma fração razoável de $i_{max} - i_{min}$, podemos então construir um *array* de rótulos, com o rótulo para o enunciado de número j na entrada da tabela com deslocamento $j - i_{min}$, com o rótulo para o *default* nas entradas não preenchidas. Para realizar o desvio múltiplo, avaliamos a expressão de forma a obtermos o valor de j, verificamos se o mesmo está no intervalo de i_{min} até i_{max}, e transferimos indiretamente para a entrada da tabela com deslocamento $j - i_{min}$. Por exemplo, se a expressão for do tipo caractere, uma tabela ligando, com 128 entradas (dependendo do conjunto de caracteres), pode ser criada e receber transferência sem nenhum teste de intervalo.

```
switch expressão
    begin
        case valor1    enunciado1
        case valor2    enunciado2
        ...
        case valorn    enunciadon
        default        enunciado0
    end
```

Fig. 8.26. Sintaxe do enunciado de desvio múltiplo.

* Seção opcional. Pode ser omitida em uma primeira leitura.

Tradução Dirigida pela Sintaxe para Enunciados de Desvios Múltiplos

Consideramos a seguinte construção de desvio múltiplo:

```
switch E
    begin
        case V1     S1
        case V2     S2
        ...
        case Vn-1   Sn-1
        default:    Sn
    end
```

Uma das opções de tradução dirigida pela sintaxe é conveniente se traduzir esse enunciado como um código intermediário equivalente ao do fig. 8.27.

Os testes todos aparecem juntos, de forma que um gerador de código simples pode reconhecer o desvio múltiplo e gerar código eficiente para o mesmo, usando a implementação mais apropriada segundo a circunstância. Se gerarmos a sequência intermediária mostrada na fig. 8.28, o compilador teria que realizar uma análise extensiva para encontrar a implementação mais eficiente. Note-se que é inconveniente se colocar os enunciados de classificação do início, porque o compilador teria então que gerar um desvio para cada um dos V's a medida que os visse.

Para traduzir na forma da fig. 8.27, introduzimos a palavra-chave **switch**, que cria dois novos rótulos, [illegible], e uma nova temporária t. Em seguida, quando analisamos a expressão E, geramos o código para avaliar E em t. Após processar E, geramos o salto [illegible].

Em seguida, ao encontrarmos cada palavra-chave **case**, criamos um novo rótulo [illegible] e o colocamos na tabela de símbolos. Colocamos então, na pilha, uma entrada para o valor V da constante associada a esse caso, e o rótulo [illegible]. Se encontrarmos o enunciado **default**, colocamos um marcador na pilha para separar as alternativas de um desvio múltiplo daquelas do desvio múltiplo mais externo.

Processamos cada enunciado **case** V : S emitindo o rótulo recém-criado [illegible], seguido pelo código para S, seguido pelo desvio [illegible]. Em seguida, quando a palavra-chave **end**, que termina o corpo do desvio múltiplo, é encontrada, estamos prontos para gerar o código da classificação. Lemos os pares valor-rótulo da pilha, criando

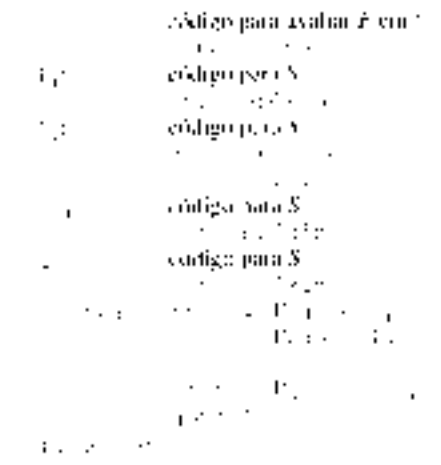


Fig. 8.27. Tradução de um enunciado de desvio múltiplo.

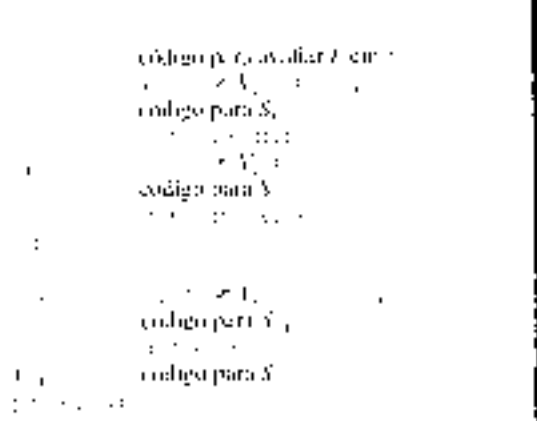
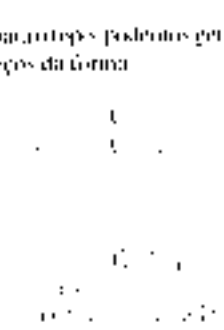


Fig. 8.28. [illegible] de um enunciado de desvio múltiplo.

sentenças desvio condicional, que provavelmente gerarão uma sequência de simples desvios de instruções da forma



onde [illegible] é o valor que abriga o valor da expressão selecionada [illegible] da rotina para o enunciado **default**. A construção de três endereços [illegible] é uma sequência para [illegible], como na fig. 8.27, mas nesse [illegible] mais fácil para o gerador de código final detectar as construções candidatas para um tratamento especial. No caso de geração de código, essas sequências de instruções [illegible] podem ser traduzidas numa ramificação de tabela de alternativas de tipo mais eficiente, dependendo de quantas existam e se os valores cabem num intervalo pequeno.

8.6 RETROCORREÇÃO*

A forma mais fácil de se implementar as construções dirigidas pela sintaxe na Seção 8.4 é em duas passagens. Primeiro constrói-se uma árvore sintática para a entrada e, em seguida, caminhamos na árvore numa ordem de busca em profundidade, computando as traduções especificadas na definição. O principal problema de se gerar código para as expressões booleanas e para o fluxo de controle numa passagem única é que, durante a recursão, não podemos saber a que rótulos para onde o controle deve ir, no momento em que os enunciados condicionais e de desvio são gerados. Podemos solucionar esse problema pela geração de uma série de enunciados de desvio com alvos vazios, mas isso leva a um número excessivo de enunciados inespecificados. Cada enunciado será colocado numa lista de enunciados de desvio cujos rótulos serão preenchidos quando o rótulo apropriado puder ser determinado. Chamamos esse preenchimento subsequente de rótulos de retrocorreção.

Nesta seção, mostraremos como a retrocorreção pode ser usada para gerar código para expressões booleanas e fluxo de controle em uma única passagem. As traduções que geramos serão bastante semelhantes àquelas na Seção 8.4, exceto pela maneira na qual geramos os rótulos. Especificamente, geramos quadruplas com alvos de quádruplas. Os rótulos serão índices para esse arranjo. Para manipular as listas de rótulos, usamos três funções:

---

*Esta seção pode ser omitida em uma primeira leitura. (N. do T.)

1. `novo_quad()` cria uma nova lista apontando somente a uma nova entrada de quadrupla, e `lista(i)` retorna uma lista apontando para a lista contendo `i`.
2. `concatena(p1, p2)` concatena as listas apontadas por `p1` e `p2` e retorna uma apontador para a lista resultante.
3. `preenche(p, i)` insere `i` como rótulo destino para cada entrada na lista apontada por `p`.

Expressões Booleanas

Consideramos agora um esquema de tradução adequado à avaliação de expressões booleanas e à geração de desvios condicionais e incondicionais. Nesse esquema, cada lista contém um não terminal `E` ou `M` na gramática a fim de associar a ação semântica adequada com a produção apropriada. Considere a produção a seguir:

```
(1)   E → E1 or M E2
(2)       | E1 and M E2
(3)       | not E1
(4)       | ( E1 )
(5)       | id1 relop id2
(6)       | true
(7)       | false
(8)   M → ε
```

Os atributos sintetizados `lista_v` e `lista_f` do não terminal `E` são usados para gerar o código de desvio para as expressões booleanas. A função `proximo_quad` retorna o endereço da próxima quadrupla a ser gerada. As quádruplas com campos destino incompletos são colocadas em listas apontadas por `E.lista_v` e `E.lista_f`, como apropriado.

As ações semânticas refletem as considerações mencionadas acima. Considere a produção `E → E1 and M E2`. Se `E1` for falso, então `E` também é falso e os endereços incompletos em `E1.lista_f` devem permanecer parte de `E.lista_f`. Se `E1` for verdadeiro, então o valor de `E` será o mesmo que o valor de `E2`. Portanto, a lista `E1.lista_v` é preenchida com o endereço inicial de `E2`. O endereço inicial de `E2` é dado pelo não terminal `M` associado a `M → ε`. Como a produção `M → ε` associamos a ação semântica

```
{ M.quad = proximo_quad() }
```

A variável `proximo_quad` retorna o endereço da próxima quadrupla a ser gerada. Esse valor será armazenado sobre a lista `E2.lista_v` quando a expressão `E1 and M E2` for reduzida. O esquema de tradução é como segue:

```
(1)  E → E1 or M E2     { preenche(E1.lista_f, M.quad);
                          E.lista_v = concatena(E1.lista_v,
                            E2.lista_v);
                          E.lista_f = E2.lista_f }
(2)  E → E1 and M E2    { preenche(E1.lista_v, M.quad);
                          E.lista_v = E2.lista_v;
                          E.lista_f = concatena
                            (E1.lista_f, E2.lista_f) }
(3)  E → not E1         { E.lista_v = E1.lista_f;
                          E.lista_f = E1.lista_v }
(4)  E → ( E1 )         { E.lista_v = E1.lista_v;
                          E.lista_f = E1.lista_f }
(5)  E → id1 relop id2  { E.lista_v = novo_lista
                            (proximo_quad);
                          E.lista_f = novo_lista
                            (proximo_quad + 1);
                          emite('if' id1.lugar relop.op
                            id2.lugar 'goto _');
                          emite('goto _') }
(6)  E → true           { E.lista_v = novo_lista
                            (proximo_quad);
                          emite('goto _') }
(7)  E → false          { E.lista_f = novo_lista
                            (proximo_quad);
                          emite('goto _') }
(8)  M → ε              { M.quad = proximo_quad }
```

Por uma questão de simplicidade, a ação semântica (5) gera duas entradas, uma desvio condicional e um incondicional. Nenhum preenchimento de destino de desvio precedeu do. O índice de proximo_quad gerado é o endereço da entrada com a lista `E.lista_v` apontando para o primeiro desses endereços gerados. _ é usado como marcador de destino para ser preenchido mais tarde com uma operação `preenche` envolvendo `E.lista_v`.

Exemplo 5.6. Considere a expressão

```
a < b or c < d and e < f
```

Uma árvore gramatical anotada é mostrada na Fig. 5.29. As ações são realizadas durante uma travessia em profundidade da árvore. Como todas as ações semânticas ocorrem no fim das regras, podem ser realizadas em conjunto com as reduções durante uma análise ascendente. Vamos supor que a resposta à redução de `a < b` pela produção (5) são as duas quádruplas

```
100: if a < b goto _
101: goto _
```

são geradas. O não terminal `M` na produção `E → E1 and M E2` registra o valor de `proximo_quad`, que é 102. A redução de `c < d` pela produção (5) gera as quádruplas

```
102: if c < d goto _
103: goto _
```

A ação para `E → E1 and M E2` faz com que `backpatch` ligue 101 a 102. A ação para `E → E1 or M E2` faz com que `backpatch` ligue 103 a 102. Essas duas ações são realizadas durante a redução de `a < b or c < d and e < f` para `E` pela produção (1). As quádruplas geradas são

```
100: if a < b goto _
101: goto 102
102: if c < d goto _
103: goto _
104: if e < f goto _
105: goto _
```

A ação semântica associada à redução final através de `E → E1 or M E2` chama `preenche(101, 103)`, o que leva a seguinte lista:

```
100: if a < b goto _
101: goto 103
102: if c < d goto _
103: goto _
104: if e < f goto _
105: goto _
```

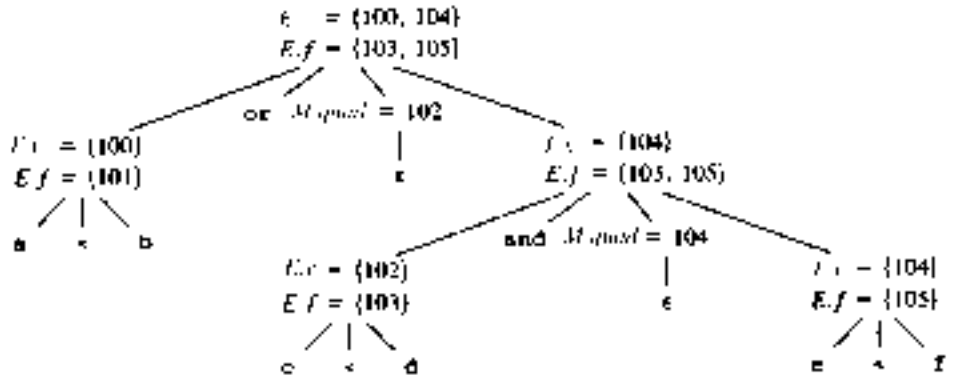


Fig. 8.29. Árvore gramatical anotada para

A expressão inteira a < b or c < d and e < f é verdadeira se a < b for verdadeira, ou se c < d for verdadeira, ou se e < f for verdadeira. Essas instruções serão satisfeitas preenchendo-se listas de desvios, quando for conveniente. Uma questão importante é a ordem em que os valores de verdade são calculados em uma expressão booleana.

Enunciados de Fluxo de Controle

Mostramos agora como a retrotradução pode ser usada para traduzir enunciados de fluxo de controle em uma só passagem. Como de costume, focalizamos nossa atenção na geração de quádruplas e a notação relacionada à tradução de testes de campos e procedimentos para o gerenciamento de listas, proveniente da seção anterior é mantida nesta seção igualmente. Como um exemplo inicial, descrevemos um esquema de tradução para enunciados gerados pela seguinte gramática:

(1)	$S \rightarrow$	**if** E **then** S
(2)		**if** E **then** S **else** S
(3)		**while** E **do** S
(4)		**begin** L **end**
(5)		A
(6)	$L \rightarrow$	L ; S
(7)		S

Aqui, S denota um enunciado, L uma lista de enunciados, A um enunciado de atribuição e E uma expressão booleana. Note-se que pode haver outras produções, tais como aquelas para os enunciados de atribuição. As produções fornecidas, contudo, serão suficientes para ilustrar as técnicas usadas para traduzir os enunciados de fluxo de controle.

Usamos as mesmas estruturas de código para os enunciados **if-then**, **if-then-else** e **while-do** que as usadas na Seção 8.6. Fazemos a suposição básica de que o código que se segue a cada enunciado em execução também segue imediatamente no vetor de quádruplas. Se tal não for verdade, um desvio explícito precisará ser primeiro gerado.

Nosso código geral será o de preencher os desvios para listas de enunciados quando seus destinos forem determinados. Não somente as expressões booleanas precisam de duas listas de desvios, que ocorrem quando uma expressão é verdadeira ou quando é falsa, mas também os enunciados, que precisam igualmente de listas de desvios dadas pelo atributo *próximo_lista* para os códigos que se seguem na sequência de execução.

Esquema para Implementar a Tradução

Descrevemos agora um esquema de tradução dirigida por sintaxe para gerar traduções para as estruturas de fluxo de controle dadas acima. O não-terminal S possui dois atributos sintetizados, *próximo_lista* e *próximo_quad*, como o L e S também o necessitam. Cada um de uma lista de quádruplas vai precisar ser eventualmente preenchida para completá-la. Essas listas são apontadas pelos atributos *próximo_lista*. Os E possuem *próximo_lista* e *próximo_quad* mais um ponteiro para uma lista de desvios condicionais e incondicionais para quádruplas que serão preenchidos quando o enunciado seguinte for conhecido e *próximo_lista* for determinado.

Na declaração do código para $S \rightarrow$ **while** E **do** S_1 na Fig. 8.30, os atributos *início* e *próximo_lista* de S são usados para marcar o início do código para o enunciado completo. E é para o corpo S_1. Vê-se as ocorrências do não-terminal M na seguinte produção, registramos os números de quádruplas a esses posições:

$$S \rightarrow \textbf{while}\ M_1\ E\ \textbf{do}\ M_2\ S_1$$

De novo, a nova produção para M, $M \rightarrow \varepsilon$, com ação estabelecendo o atributo *M.quad* como o número da próxima quádrupla. Assim, o corpo S_1 do enunciado *while* será executado sempre que o controle fluir para o início. Precisamos também preencher os **while** M_1 **do** M_2 S_1 em contrapartida S *próximo_lista* de forma a fazer com que todos os alvos sejam avaliados após o código para S, porque o controle pode também "cair através do final". $E.falso$ é retrocorrigido de forma a ir para o início de S fazendo-se esses desvios em $E.falso$ iguais para $M_1.quad$.

Uma alternativa mais forte para se usar *S próximo_lista* e *L próximo_lista* vem quando o código é gerado para o enunciado condicional **if** E **then** S_1 **else** S_2. Se o controle "cai através do fim" de S_2, como quando S_2 é uma atribuição, precisamos incluir no final do código para S_2 um salto sobre o código para S_1. Usamos um marcador para atribuir esse salto após S_2. Seja A um não-terminal com a produção $A \rightarrow \varepsilon$. A possui atributo *A.próximo_lista*, que é igual a uma lista contendo um único desvio incondicional que é gerado pela regra semântica para A. Introduzimos agora as ações para a gramática revisada

$$\begin{aligned} (1)\quad S \rightarrow\ & \textbf{if}\ E\ \textbf{then}\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2 \\ & \{\textit{retrocorrigir}(E.\textit{falso}, M_2.\textit{quad}); \\ & \textit{retrocorrigir}(E.\textit{listat}, M_1.\textit{quad}); \\ & S.\textit{próximo\_lista} := \textit{concatenar}(S_1.\textit{próximo\_lista}, \\ & \quad \textit{concatenar}(N.\textit{próximo\_lista}, S_2.\textit{próximo\_lista}))\} \end{aligned}$$

Retrocorrigimos os valores quando E for verdadeiro para saltar para $M_1.quad$, que é o início do código para S_1. Similarmente, retrocorrigimos os desvios quando E for falso de forma que o controle salte para o início do código para S_2. A lista $S.próximo\_lista$ contém todos os saltos para fora de S_1 e de S_2, bem como o desvio gerado por N.

$E.V := \epsilon$	{ *V.proxima_lista* = *concatena(V.proxima_lista, endereço)*; *emite('*$\_$*', '*$\_$*')* }
$M \rightarrow \epsilon$	{ *M.quad* = *proximo_quadruplo* }
$N \rightarrow$ **if** B **then** $M\ S$	{ *retrocede(B.lista_V, M.quad)*; *S.proxima_lista* = *concatena(B.lista_F, S.proxima_lista)* }
$S \rightarrow$ **while** $M_1\ E$ **do** $M_2\ S$	{ *retrocede(S.proxima_lista, M1.quad)*; *retrocede(E.lista_V, M2.quad)*; *S.proxima_lista* = *E.lista_F*; *emite('*$\_$*', '*$\_$*', M1.quad)* }
$S \rightarrow$ **begin** L **end**	{ *S.proxima_lista* = *L.proxima_lista* }
$S \rightarrow A$	{ *S.proxima_lista* = **nil** }

Observação: *V.proxima_lista* = **nil** significa *V.proxima_lista* aponta para *Nada*

$M \rightarrow L,\ M\ S$	{ *retrocede(L.proxima_lista, proximo_lista)*; *M.quad* = *L.proximo_lista* }

Depois disso, o que se segue a L na ordem de execução é o início de S. Isso é conseguido fazendo a lista *L.proxima_lista* ser retrocorrigida para conter o endereço do início de S, que é dado por *M.quad*.

$N \rightarrow \epsilon$	{ *N.proxima_lista* = *concatena(N.proxima_lista, endereço)*; *emite('*$\_$*', '*$\_$*')* }

Note-se que as regras gerais dessas quatro últimas regras são quase idênticas às regras semânticas, a não ser pelas regras 21 e 31. Todas estas regras geram código para as expressões booleanas associadas aos comandos de decisão e de repetição e expressões. O que é preciso garantir é que o fluxo de controle faz o programa objeto ter uma ordem adequada, de forma que as avaliações das variáveis booleanas sejam executadas adequadamente.

Rótulos e Desvios

A construção de linguagem de programação mais elementar para modificar o fluxo de controle de um programa é constituída pelos rótulos e desvios. Quando um compilador encontra um comando como

```
goto rotulo
```

precisa ter ficado sabendo anteriormente um endereço de memória para o escopo desse comando de desvio. Se o rótulo já tiver aparecido, certamente então não deve haver uma declaração de rótulo associada a esse rótulo (pelo menos não em uma linguagem bem projetada), e a tabela de símbolos já terá uma entrada fornecendo o rótulo gerado pelo compilador para a primeira instrução de três endereços associada ao comando `goto rotulo`. Para a tradução, geramos um comando de três endereços `goto` _ tendo aquele rótulo gerado pelo compilador como destino.

Quando um rótulo `rotulo` é encontrado pela primeira vez no programa fonte, quer numa declaração quer como alvo de um desvio `goto rotulo`, um rótulo `rotulo` na tabela de símbolos é gerado, e o rótulo simbólico para `rotulo` é também gerado. Esse rótulo é colocado em um campo de execução que contém as informações sobre os parâmetros para os procedimentos, e as chamadas e retornos são parte do pacote de suporte em tempo de execução. Discutimos os diferentes tipos de mecanismos necessários para implementar o pacote de suporte em tempo de execução no Capítulo 7. Nesta seção, discutimos as estratégias que são tipicamente usadas para chamadas e retornos de procedimentos.

Vamos considerar uma gramática para um comando simples de chamada de procedimento.

(1) $S \rightarrow$ **call id** $(Elist)$	
(2) $Elist \rightarrow Elist,\ E$	
(3) $Elist \rightarrow E$	

Seqüências de Chamada

Como discutido no Capítulo 7, a tradução de uma chamada inclui uma seqüência de chamada. A seqüência de ações tomadas quando uma chamada de procedimento ocorre é importante para a implementação de variáveis locais e não-locais. Quando as seqüências de chamada diferem, nestes aspectos, para implementações distintas de uma mesma linguagem, as seguintes ações tipicamente ocorrem.

Quando uma chamada de procedimento ocorre, é necessário inserir informação para o registro de ativação do procedimento chamado. Os argumentos do procedimento chamado precisam ser avaliados e tornados disponíveis para uso pelo procedimento chamado. Os apontadores de ambientes precisam ser estabelecidos de forma a ter acesso pelo procedimento chamado a dados não-locais nos blocos envolventes. O estado do procedimento chamador precisa ser salvo de forma que ele possa prosseguir a execução após a chamada. O endereço de retorno também é, em geral, localizado. A localização para a qual a rotina chamada precisa transferir o controle de volta após ter completado. O endereço de retorno é usualmente a localização de uma instrução que segue a chamada no procedimento chamador. Finalmente, os valores para início do código no procedimento chamado precisam ser gerados.

Quando o procedimento retorna, várias ações precisam ocorrer. Se o procedimento chamado for uma função, o resultado precisa ser armazenado em local conhecido. O registro de ativação do procedimento chamador pode ser restaurado. Um salto para o endereço de retorno do procedimento chamador precisa ser gerado.

Não existe divisão exata de tarefas entre o procedimento chamador e o chamado. Frequentemente, a linguagem fonte, a máquina-alvo e o sistema operacional impõem exigências que favorecem uma solução em relação a outra.

Um Exemplo Simples

Vamos considerar um exemplo simples no qual os parâmetros são transmitidos por referência e a memória é dividida estaticamente. Nesta situação, podemos usar os próprios endereços `id.lugar` como os endereços de l-valor para os argumentos. No procedimento chamado, é passado um registrador contendo um apontador para o primeiro dos endereços `id.lugar`, e o procedimento chamado pode então obter os apontadores para quaisquer de seus argumentos através do uso de deslocamentos adequados a partir desse apontador inicial. A seguinte código de três endereços para essa tipo de chamada é suficiente para explicitar os valores de três atributos que existem para avaliar aqueles argumentos que sejam expressões que não nomes simples. Eles são `E.lugar`, um para cada argumento. Se não desejamos misturar os endereços que avaliam os parâmetros com os endereços `id.lugar`, teremos que salvar o valor de *l-valor* para cada expressão E existente na lista $(E, E, \ldots, E)$.[1]

[1] Outra estratégia seria avaliar todos os argumentos e colocar seus valores em registradores ou na pilha de execução, onde eles poderiam ser encontrados pelo procedimento chamado. Discutimos essa estratégia no Capítulo 7.

[1] Note que o termo *l-valor* é usado para qualquer endereço de memória, enquanto *r-valor* é um valor que pode ser colocado em um registrador. Os termos são derivados da posição que eles ocupam em uma atribuição. Uma expressão pode gerar um endereço de memória, que pode ser usado para recuperar um valor ou para armazenar um valor.

uma estrutura de dados conveniente na qual podemos salvar pares de símbolo e valor, uma lista LIFO (primeiro a entrar, primeiro a sair). Nessa forma sintática para $bind\_E \rightarrow bind\_E, E$ marca um par entre símbolo e valor semântico para $S \rightarrow$ **call id** $bind\_E$ gera um procedimento que faz cada um dos parâmetros ser avaliado e salvo, e então chama o procedimento. Ações semânticas para $E \rightarrow$ **id** $bind\_E$ geram código para encontrar o valor mais recente salvo para **id** e colocá-lo na pilha.

a) Dê o código para $S \rightarrow$ **call id** $bind\_E$.

b) Dê o código para $bind\_E$, que avalia cada uma das expressões, empilha os valores e, depois, chama o procedimento apontado pelo argumento, seguido por um número de chamada de procedimento. Note que a mesma variável pode ser usada em mais de uma expressão. Você pode gerar uma contagem do número de parâmetros ou armazenar um marcador. Cada valor armazenado na pilha deve ser acompanhado de um valor que identifique a variável, mas a mesma palavra pode ser calculada da forma que você preferir.

c) Dê o código para $E \rightarrow bind\_E$ **id**.

d) Dê o código para $bind\_E \rightarrow \varepsilon$.

Agora, quando você usar **id**, o código deve empilhar o valor apropriado para a localização, ou de valor-l, do símbolo para o nome que denota o valor de E.

EXERCÍCIOS

8.1 Traduza a expressão [illegible] em cada uma das seguintes formas:

a) Uma árvore sintática.

b) Notação pós-fixa.

c) Código de três endereços.

8.2 Traduza a expressão [illegible] em

a) Quádruplas.

b) Tuplas.

c) Tuplas indiretas.

8.3 Traduza em cada um dos exercícios a seguir o programa C

[illegible]

em

a) Uma árvore sintática.

b) Notação pós-fixa.

c) Código de três endereços.

***8.4** Prove que se todos os operadores são binários, uma cadeia de operadores e operandos é uma expressão pós-fixa se e somente se (1) existe exatamente um operador a menos do que o número de operandos e (2) cada prefixo não vazio da expressão possui menos operadores do que operandos.

8.5 Modifique o esquema de tradução da Fig. 8.11 que computa os pesos e endereços relativos dos nomes de registros de forma a permitir listas de nomes em lugar de nomes isolados em declarações da forma $D \rightarrow$ **id** T.

8.6 A forma *pós-fixada* de uma expressão na qual o operador θ é aplicado a expressões $e_1, e_2, \ldots, e_n$ é $p_1 p_2 \ldots p_n \theta$, onde p_i é a forma pós-fixada de e_i.

a) Dê a forma pós-fixada de $(a + b) * (c + d) / (e + f)$.

**b)* Mostre que as expressões infixas não podem ser traduzidas na forma pós-fixa por esquemas de tradução orientados aos sintetizados, a menos que atributos herdados ou marcadores sejam introduzidos.

c) Forneça uma definição dirigida pela sintaxe para traduzir expressões infixas na forma pós-fixa. Quais dos métodos do Capítulo 5 você pode usar?

8.7 Escreva um programa para implementar a definição dirigida pela sintaxe para a tradução de expressões booleanas em código de três endereços, dada na Fig. 8.24.

8.8 Modifique a definição dirigida pela sintaxe da Fig. 8.24 para que gere código para a máquina de pilha da Seção 2.8.

8.9 A definição dirigida pela sintaxe da Fig. 8.2 traduz $A \rightarrow$ **id**, **id**, em pares de comandos

```
if id, < int goto
goto ...
```

Precedemos-os através, traduzindo-os como um único comando

```
if id, < id, goto ...
```

e seguimos cada fronteira pelo código para A quando é traduzida verdadeiramente. Modifique a declaração na Fig. 8.24 para gerar o código desse jeito.

8.10 Escreva um programa para implementar a definição dirigida pela sintaxe da Fig. 8.33 para os comandos de fluxo de controle.

8.11 Escreva um programa para implementar o algoritmo de reconhecimento de duplos da Seção 8.6.

8.12 Traduza o seguinte comando de atribuição em código de três endereços usando o esquema de tradução da Seção 8.5.

[illegible]

***8.13** Algumas linguagens, tais como PL/I, permitem que a uma lista de nomes seja fornecida uma lista de atributos e também que as declarações sejam aninhadas umas nas outras. O seguinte programa abstrato ilustra o problema:

$$
\begin{aligned}
D &\rightarrow \textit{lista\_de\_nomes}\ \textit{lista\_de\_atributos} \\
&\quad (D)\ \textit{lista\_de\_atributos} \\
\textit{lista\_de\_nomes} &\rightarrow \textbf{id}, \textit{lista\_de\_nomes} \\
&\quad \textbf{id} \\
\textit{lista\_de\_atributos} &\rightarrow ,\ \textit{lista\_de\_atributos} \\
&\quad A \\
A &\rightarrow \textbf{decimal}\ \ \textbf{fixed}\ \ \textbf{float}\ \ \textbf{real}
\end{aligned}
$$

O significado de $D \rightarrow (D)$ *lista_de_atributos* é que a todos os nomes nomeados na declaração entre parênteses separados por vírgulas os atributos em *lista_de_atributos*, além daqueles que eles já possuem, são adicionados. Note que uma declaração de nomes e atributos pode fazer com que vários itens de informação sejam colocados na tabela de símbolos. Forneça uma definição dirigida pela sintaxe para as declarações desta gramática.

8.14 Em C++ um comando [illegible] possui a seguinte forma

[illegible]

Tomando o seguinte subprograma:

```
f(
        x,
        y,
        z)
    real x
    int y
    ptr z
{
```

construa uma definição dirigida pela sintaxe para traduzir cada um dos comandos de estilo de C em código de três endereços.

8.15 O Pascal padrão define o comando

```
for v := valor_inicial to valor_final do corpo
```

como tendo o mesmo significado que a sequência de comandos de código

```
início
    t1 := valor_inicial; t2 := valor_final;
    se t1 <= t2 então início
        v := t1;
        corpo;
        enquanto v <> t2 faça início
            v := succ(v);
            corpo
        fim
    fim
fim
```

a) Considere o seguinte programa Pascal:

```
program p(input, output);
  var i, j, k: integer;
  procedure q(m, n: integer);
    var i, j: integer;
    begin
      i := m + 1;
      j := n + 2;
      k := i + j
    end;
  begin
    i := 1;
    j := 2;
    k := 3;
    q(i + j, k);
    writeln(i, j, k)
  end.
```

Que comportamento esse programa teria se o passo de cópia fosse substituído por passagem por referência? Se passagem por nome, onde i + j é avaliado na ativação de q?

b) Construa uma definição dirigida pela sintaxe que gere o código de três endereços correto para os encadeamentos em Pascal.

NOTAS BIBLIOGRÁFICAS

UNCOL (para Universal Computer Oriented Language — Linguagem Universal Orientada para Computadores) é uma linguagem intermediária que tem despertado interesse desde meados da década de 50. Uma forma UNCOL é relatada em comitê por Strong et al. [1958], mostrando como os compiladores poderiam ser construídos portando-se para cada nova máquina da para uma dada linguagem-fonte com uma interface de retargetação para cada dada linguagem-fonte. Assim, os compiladores projetados por relatores são relativamente usados para vender vários compiladores para novas máquinas. Steel [1961] e Sayre [1969] também propuseram UNCOL.

Um compilador incremental consiste em um gerador de cargas que pode ser escrito de forma portável com várias interfaces de retargetação que implementam uma dada linguagem em várias máquinas. Nesse é um exemplo primordial de linguagem com um compilador incremental (Huskey, Halstead e McArthur [1960]) escrito em sua própria linguagem. Ver também Richards [1971], para uma descrição de um compilador incremental para BCPL, Nori et al. [1981] para Pascal P e Johnson [1978] para C. Newey, Poole e Waite [1972] aplicam a ideia de se usar uma interface de retargetação para uma máquina abstrata orientada a pilha de forma a compilar um compilador fácil.

Hopgood [1969], de implementação de linguagens em máquinas escrevendo interfaces de retargetação e interfaces de retargetação explorando as semelhanças entre compiladores distintos com uma abordagem de diversas maneiras. Uma dessas abordagens é se adaptar uma interface de retargetação para uma nova linguagem sobre um compilador existente. Feldman [1979b] descreve a adição de uma interface de retargetação para Fortran 77 aos compiladores C, por Johnson [1979] e Ritchie [1979]. Organizações de compiladores projetados para aceitar múltiplas interfaces de retargetação e de retargetação são descritas por Davidson e Fraser [1984b], Love et al. [1981] e Tanenbaum et al. [1983].

Os termos "tamanho" e "interesse" de máquinas abstratas usados no Davidson e Fraser [1984b] destacam o papel do conjunto de operadores permitidos uma representação intermediária. O conjunto de instruções e modos de endereçamento de uma máquina-alvo são limitados e, por conseguinte, as interfaces de retargetação não têm que fazer muitas escolhas ao gerar o código objeto. As máquinas-alvo não provêm alternativas de implementação de construções em nível de código-fonte. Uma vez que todas as alternativas podem ser implementadas facilmente por todas as máquinas-alvo e o compilador-fonte não pode permitir uma dependência da máquina-alvo, de forma que se consegue um zero crescente a partir das interfaces similares se aplicam a construções de código intermediário. Tais construções simplificam o código de três endereços. Fraser e Hanson [1982] consideram formas de avaliar o acesso à pilha em tempo de execução usando operações independentes da máquina.

A implementação de A-ports é discutida em detalhes por Ramsdell e Russell [1961] e Gram, Holt e Langmaack [1969]. Freudenberg [1968] discute PEG. Wirth [1971], Pascal-P e Ammann et al. [1976], Miller [1981] e Nori et al. [1981] descrevem o Pascal-P.

Mueller e Mueller [1980] e Giegerich e Wilhelm [1978] discutem a geração de código de uma para expressões booleanas. O Exercício 8.15 é proveniente de Newey e Waite [1985].

Capítulo 9

GERAÇÃO DE CÓDIGO

Vimos como um nosso modelo de compilador é o gerador de código. Recebe como entrada a representação intermediária do programa-fonte e produz como saída um programa alvo equivalente, como mostra a Fig. 9.1. As técnicas de geração de código apresentadas neste capítulo podem ser usadas logo ou após uma fase de otimização antes da geração de código, como vimos acerca dos compiladores otimizadores. O que tal fase pode transformar o código intermediário numa forma a partir da qual um código alvo mais eficiente possa ser produzido. Tais objetivos e aspectos da otimização de código serão detalhados no próximo capítulo.

As exigências tradicionalmente impostas a um gerador de código são severas. O código gerado deve ser correto e de alta qualidade, significando que o mesmo deve tomar efetivo uso dos recursos da máquina alvo. Sobretudo, o próprio gerador de código deve rodar eficientemente.

Matematicamente o problema de se gerar um código ótimo não pode ser solucionado. Na prática, devemos nos contentar com técnicas heurísticas que gerem um código bom, mas não necessariamente ótimo. A escolha dos métodos heurísticos é importante, na medida em que um algoritmo de geração de código cuidadosamente projetado pode produzir um código que seja, às vezes, mais rápido do que aquele produzido por um algoritmo concebido às pressas.

9.1 TEMAS NO PROJETO DE UM GERADOR DE CÓDIGO

Em muitos aspectos, ele é dependente do sistema operacional, tendo como, por exemplo, a gerência de memória, seleção de instruções, alocação de registradores e a ordem de avaliação sendo intervenientes a quase todos os problemas de geração de código. Nesta seção, veremos os temas genéricos do projeto de geradores de código.

Entrada para o Gerador de Código

A entrada para o gerador de código consiste na representação intermediária do programa-fonte produzida pela varredura do compilador juntamente com as informações na tabela de símbolos, que são usadas para determinar os endereços, em tempo de execução, dos objetos de dados denotados pelos nomes na representação intermediária.

Como mencionamos no capítulo anterior, existem várias escolhas para as linguagens intermediárias, incluindo as representações lineares, como a notação pós-fixa, representações de três endereços, como os quádruplos, representações de máquina virtual, como o código de máquina de pilha, e as representações gráficas, como as árvores sintáticas e os GADAs. Apesar dos algoritmos deste capítulo serem ilustrados em termos de código de três endereços, produzido a GADAs, muitas das técnicas também se aplicam a outras representações intermediárias.

Assumimos que a geração prévia de código, a partir da varredura do compilador, analisou a sintaxe do programa-fonte, bem como o traduziu numa forma razoavelmente detalhada de representação intermediária, de forma que os nomes que figuram na linguagem intermediária possam ser representados por quádruplos que a máquina alvo possa diretamente manipular (bits, inteiros, reais, apontadores etc.). Também assumimos que a necessária verificação de tipos já tenha tido lugar, de forma que os operadores de conversão de tipo já tenham sido inseridos onde quer que fossem necessários e que os erros semânticos óbvios (por exemplo, tentar indexar um vetor através de um número em ponto-flutuante) já tenham sido detectados. A fase de geração de código pode, por conseguinte, prosseguir na suposição de que a sua entrada está isenta de erros. Em alguns compiladores, esse tipo de verificação semântica é feito junto com a geração de código.

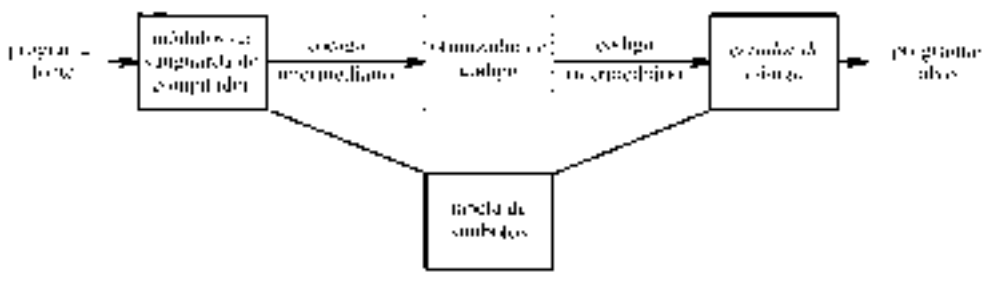


Fig. 9.1 Posição do gerador de código

Programas-Alvo

A saída do gerador de código é o programa-alvo. Os programas-alvo mais comuns são: código absoluto de máquina, código relocável de máquina ou linguagem de montagem.

Como saída a produção de um programa em linguagem absoluta tem a vantagem de que o programa pode ser colocado numa localização fixa de memória e executado imediatamente. Um pequeno programa pode ser compilado e imediatamente executado. Um certo número de compiladores para "trabalhos de estudantes", tais como WATFIV e PL/C, produz código absoluto.

A produção de um programa em linguagem relocável (ou algum módulo objeto) como saída permite que os subprogramas sejam compilados separadamente. Um conjunto de módulos objeto relocáveis pode ser ligado e carregado por um pré-processador de ligações. Apesar de termos de passar o código adicional de editar as ligações, carregar o programa e produzir módulos objeto relocáveis, ganhamos muito ao poder compilar os subprogramas separadamente e chamar outros programas previamente compilados a partir de um módulo objeto. Se a máquina-alvo não trata a relocação automaticamente, o compilador precisa providenciar informações explícitas para o carregador a fim de ligar os segmentos de programa compilados separadamente.

A produção de um programa em linguagem de montagem como saída torna o processo de geração de código um pouco mais fácil. Podemos gerar instruções simbólicas e usar facilidades de processamento de macros do montador para auxiliar na geração de código. O processo global ainda passa pelo montador após a geração de código. Como a produção do código de montagem não duplica toda a tarefa do compilador, essa escolha é uma alternativa atraente, especialmente para uma máquina com memória pequena, onde o compilador precisa realizar diversos passos. Neste capítulo, usamos o código de montagem como a linguagem-alvo, pois, quando as questões de legibilidade, correção, podem ser estudadas a partir dos deslocamentos e de strings simbólicas na tabela de símbolos, a geração de código pode produzir endereços relocáveis tão facilmente quanto endereços absolutos.

Gerenciamento de Memória

O mapeamento dos nomes nos programas-fonte para os endereços dos objetos de dados em tempo de execução é feito cooperativamente pelo compilador, pelo pacote de código e pelo sistema operacional-alvo. No último capítulo, associamos um número inteiro a cada um dos endereços sucessivos alocados para os nomes na tabela de símbolos. Na Seção 8.7, as entradas na tabela de símbolos eram usadas a medida que as declarações eram examinadas. O programa declaração determina a largura, isto é, a quantidade de memória necessária para o nome declarado. A partir das informações da tabela de símbolos, pode ser determinado um endereço relativo para o nome na área de dados para o procedimento. Na Seção 9.3, definimos as implementações da alocação estática de pilha para áreas de dados e mostramos como os nomes na representação intermediária podem ser convertidos em endereços no código-alvo.

Se deve ser gerado código de máquina, os rótulos dos quádruplos e os três endereços têm que ser convertidos para endereços de instruções. Essas idéias são análogas à técnica de "remendação" da Seção 8.6. Suponhamos que os rótulos se referem a números de quadruplas nas matrizes de quadruplas. A medida que examinamos cada quadrupla podemos deduzir a localização do primeiro instrução de máquina gerada para aquela quadrupla, mantendo uma contagem do número de palavras usadas pelas instruções geradas até então. A contagem pode ser mantida no vetor de quadruplas (cada um tem um campo extra de forma que se for encontrada uma referência futura [illegible] do quadruplo, podemos simplesmente gerar uma instrução de desvio com o endereço). Por agora, a localização de máquina da primeira instrução no código para a quadrupla i. Se, subsequentemente, o desvio para i for encontrado, podemos gerar uma instrução de desvio para i. Uma lista para quadrupla i localizará a primeira instrução gerada para a quadrupla i. Então, ao processar a quadrupla i, preenchemos a localização adequada em todas as instruções que se referem a i anteriormente.

Seleção de Instruções

A natureza do conjunto de instruções da máquina-alvo determina a dificuldade da seleção de instruções. A uniformidade e completeza do conjunto de instruções são fatores importantes. Se a máquina-alvo não suporta cada tipo de dado de uma maneira uniforme, cada exceção à regra geral requer um tratamento especial.

A velocidade das instruções e os dialetos de máquina são fatores importantes. Se o único requisito for uma eficiência razoável, a geração de instruções é um processo direto. Para cada tipo de instrução de três endereços, podemos projetar um esqueleto de código que descreve o código-alvo a ser gerado para aquela construção. Por exemplo, cada instrução de três endereços da forma [illegible] pode ser traduzida na seqüência de código

[illegible]

Infelizmente, esse tipo de geração de código engendrado é muito frequentemente produz um código de baixa qualidade. Por exemplo, a seqüência de instruções

[illegible]

será traduzida em

[illegible]

Aqui, o quarto enunciado é redundante, assim também o será o terceiro, caso o próximo enunciado seja rotulado subsequentemente.

A qualidade do código gerado é determinada pela sua velocidade e tamanho. Uma máquina-alvo com um rico conjunto de instruções pode prover diversas formas de se implementar a dada operação. Uma vez que as diferenças de custo entre as diferentes implementações podem ser significativas, uma tradução ingênua do código intermediário pode levar a um código-alvo correto, porém muito ineficiente. Por exemplo, se a máquina-alvo possui uma instrução de "incremento" [illegible], o enunciado de três endereços [illegible] pode ser implementado mais eficientemente pela instrução sozinha [illegible] do que por uma seqüência mais óbvia que carregue [illegible] em um registrador, adicione uma constante e, em seguida, armazene o resultado de volta em [illegible].

[illegible]

As características das instruções são necessárias para se projetar boas seqüências de código, mas infelizmente, informações acuradas a respeito das características das instruções são frequentemente difíceis de se obter. Decidir que seqüência de código de máquina é a melhor para uma dada construção de três endereços pode também envolver

respeito do contexto em que a instrução aparece. As transformações para se construir os sobretempos de instruções são discutidas na Seção 9.12.

Alocação de Registradores

As instruções envolvendo operadores do tipo registrador são, geralmente, mais rápidas do que aquelas envolvendo operandos na memória. Por conseguinte, a utilização eficiente dos registradores é particularmente importante na geração de código de uma qualidade razoável. Uso de registradores é frequentemente subdividido em dois subproblemas:

1. Durante a alocação de registradores, se selecionam o conjunto de variáveis que residirão em registradores a um determinado ponto do programa.
2. Durante a fase subsequente de *atribuição de registradores*, selecionam-se os registradores específicos nos quais a variável residirá.

Encontrar uma atribuição ótima para os registradores é difícil, caso se queira um número mínimo de registradores. Matematicamente, o problema é NP-completo.[1] O problema é ainda mais complicado porque o hardware ou o sistema operacional podem exigir que certas convenções de uso dos registradores sejam observadas.

Certas máquinas requerem pares de registradores. Uma instrução de dois endereços pode usar o registrador seguinte de número ímpar para dar alguns operandos resultados. Por exemplo, nas máquinas do System/370 da IBM[2], a multiplicação e a divisão envolvem pares de registradores. A instrução de multiplicação da forma

```
m   r,x
```

onde `r` é o número de um registrador par, multiplica o conteúdo do registrador `r+1` pelo conteúdo de `x`. O valor do produto ocupa o registrador ímpar `r+1`, e o conteúdo do registrador `r` é ignorado. O produto ocupa, portanto, o par de registradores.

A instrução de divisão da forma

```
d   r,x
```

onde `r` é o dividendo de 64 bits ocupando o par `r`, `r+1` de registradores, onde `r` é um registrador par, e `x` representa o divisor. Após a divisão, o registrador par `r` contém o resto e o ímpar `r+1` o quociente.

Nas Figs. 9.2(a) e (b), mostram-se duas sequências de código de três endereços na Fig. 9.2(a) e (b), na qual a única diferença é o operando do segundo enunciado. As sequências de código de montagem para (a) e (b) são fornecidas na Fig. 9.3.

[illegible] registrador `[illegible]` deslocado de valendo para `[illegible]` de forma que todos os seis bits fiquem iguais ao sinal (isto é, `[illegible]` ou `[illegible]`). Os quatro bits de ordem mais baixa de `[illegible]` contêm o dividendo. Após a divisão, o registrador `[illegible]` contém o resto e `[illegible]` o quociente.

Note-se que a escolha ótima para o registrador no qual `[illegible]` deve residir depende do que acontecerá a seguir (uma informação global). As estratégias para a alocação de registradores são discutidas na Seção 9.6.

Escolha da Ordem de Avaliação

A ordem na qual as computações são realizadas pode afetar a eficiência do código gerado. Algumas computadoras requerem que os registradores usados para dois operandos de uma operação sejam contíguos. Por exemplo, a multiplicação em ponto flutuante do [illegible] requer que o multiplicando esteja em um registrador par e o multiplicador no registrador ímpar imediatamente seguinte. Em geral, é difícil encontrar a ordem ótima (ou mesmo uma boa ordem) para as computações, mesmo quando a ordem em que as expressões são geradas é especificada. Mais adiante, neste capítulo, discutiremos algoritmos que geram código para várias expressões em uma ordem que é frequentemente a melhor possível.

```
t := a + b
t := t * c
t := t / d
```

(a)

```
t := a + b
t := t + c
t := t / d
```

(b)

Fig. 9.2 Duas sequências de código de três endereços

Infelizmente, existem casos importantes para os quais a geração de código que produza um código correto é difícil. A correção ganha a gratificante recompensa por causa do número de casos especiais com que o gerador de código pode se deparar. Dado o prêmio da correção, o projeto de um gerador de código de tempo hábil pode ser, frequentemente, uma experiência frustrante e demorada, uma importante fonte de prazer.

A Seção 9.6 contém uma algoritmo direto para a geração de código que usa a informação sobre os usos subsequentes de um operando para gerar código para uma máquina de registradores. O algoritmo considera uma instrução por vez, mantendo os operandos em registradores tanto quanto possível. A saída do gerador de código pode ser melhorada através de técnicas de otimização específicas, como aquelas discutidas na Seção 9.9.

A Seção 9.8 apresenta técnicas para fazer um uso bom dos registradores através de um bloco básico. Um bloco básico é uma unidade de código intermediário. Vê-se que está associação de registradores para operandos é, frequentemente, adequada em blocos inteiros.

As Seções 9.10 e 9.11 apresentam a gama de técnicas de geração de código utilizadas para várias construções de geradores de código sofisticados. Várias versões do PCC — o compilador C portável com um gerador de código bem instalado em muitas máquinas — estão disponíveis sob licença do UNIX. Devido à variedade de máquinas sendo alvo em muitas instalações do PCC, a Seção 9.12, mostra como a geração de código pode ser tratada como um processo de reescrita.

9.2 A MÁQUINA-ALVO

Uma familiaridade com a máquina alvo e seu conjunto de instruções é um pré-requisito para o projeto de um bom gerador de código. Infelizmente, uma discussão geral da geração de código não pode considerar os detalhes de qualquer máquina alvo com exatidão suficiente para formar apto a gerar um código de boa qualidade para uma linguagem complexa naquela máquina. Neste capítulo, usaremos como computador-alvo uma máquina de registradores que é representativa para vários minicomputadores. Nas [illegible] técnicas de geração de

```
L    R1, a
A    R1, b
M    R0, c
D    R0, d
ST   R1, t
```

(a)

```
L    R0, a
A    R0, b
A    R0, c
SRDA R0, 32
D    R0, d
ST   R1, t
```

(b)

Fig. 9.3 Sequências de instruções para (a) e (b)

---

[1] Para uma discussão sobre problemas NP-completos, veja Aho, Hopcroft e Ullman [1974]. Uma prova de que a alocação de registradores é difícil pode ser encontrada em Sethi [1973]. Veja também Horwitz, Karp, Miller e Winograd [1966] para uma discussão sobre problemas de alocação de registradores.

[2] IBM System/370 Principles of Operation, IBM Publication Number GA22-7000, IBM Corporation, Poughkeepsie, New York.

código apresentados neste capítulo também foram usados em muitas outras classes de máquinas.

Nosso computador alvo é uma máquina endereçável em nível de byte, com quatro bytes por palavra, e 8 registradores de propósito geral [illegible]. Possui instruções de três endereços da forma

[illegible]

em que tanto o código da operação e os registradores quanto os campos de endereço são campos definidos. A máquina possui os seguintes códigos de operação de interesse:

[illegible]	copia o valor para [illegible]
[illegible]	calcula [illegible] e coloca o resultado em [illegible]
[illegible]	subtrai [illegible] de [illegible]

Outras instruções serão introduzidas na medida do necessário.

Os campos origem e destino não são largos o suficiente para abrigar todos os endereços de memória, e, dessa forma, determinados padrões nesses campos especiais indicam que as palavras seguintes à [illegible] instrução contêm operandos ou endereços. A origem e o destino de uma instrução são especificados pela combinação de registradores e [illegible] de memória correspondentes a cada endereçamento. Cada seção que se segue começa com a sintaxe do modo de endereçamento em questão, [illegible] a descrição é apresentada para

Os modos de endereçamento, juntamente com suas formas na linguagem de montagem e custos associados, são como se segue.

Modo	Forma	Localização	Custo Adicional
absoluto	[illegible]	[illegible]	1
registrador	[illegible]	[illegible]	0
indexado	[illegible]	[illegible] = *conteúdo* [illegible]	1
registrador indireto	[illegible]	*conteúdo* [illegible]	0
indexado indireto	[illegible]	*conteúdo* [illegible] + *conteúdo* [illegible]	1

Uma localização de memória [illegible] ou um registrador [illegible] representados da mesma maneira quando usados como origem ou destino. Por exemplo, a instrução

[illegible]

armazena o conteúdo do registrador [illegible] na localização de memória [illegible].

Um deslocamento de endereço a partir do valor no registrador [illegible] é escrito [illegible]. Por conseguinte,

[illegible]

armazena o valor

conteúdo [illegible] + *conteúdo* [illegible]

na localização de memória [illegible].

As versões indiretas dos últimos dois modos de endereçamento são indicadas pelo prefixo [illegible]. Por conseguinte

[illegible]

armazena o valor

conteúdo de conteúdo [illegible] + *conteúdo* [illegible]

na localização de memória [illegible].

Um modo de localização adicional permite que a origem seja uma constante:

Modo	Forma	Localização	Custo Adicional
literal	[illegible]	[illegible]	1

Por conseguinte, a instrução

[illegible]

sempre coloca a constante 1 no registrador [illegible].

Custos das Instruções

Fazemos o custo de uma instrução ser um mais os custos associados aos modos de endereçamento daqueles operandos que têm custos adicionais. Isso é uma medida razoável do comprimento e velocidade de execução da instrução. Os modos de endereçamento envolvendo registradores somente são rápidos, enquanto aqueles envolvendo a localização de memória ou literal possuem custo mais alto porque tais operandos precisam ser armazenados junto com a instrução.

Se o espaço for importante, fazemos de maneira que o custo meça o comprimento das instruções. Normalmente, isso assegura que os modos mais rápidos sejam os mais curtos. Para ilustrar alguns exemplos, o custo pretende ser uma medida do comprimento da instrução e também do tempo gasto executando-se a instrução. Por conseguinte, através da minimização do custo, tentamos minimizar o comprimento e o tempo gasto executando a instrução. Por vezes, há tendências opostas, mas normalmente é possível otimizar o tempo gasto para executar a instrução igualmente. Alguns exemplos seguem.

1. A instrução [illegible] copia o conteúdo do registrador [illegible] no registrador [illegible]. Essa instrução possui custo um, pois ocupa somente uma palavra de memória.

2. A instrução de armazenamento [illegible] copia o conteúdo do registrador [illegible] na localização de memória [illegible]. Essa instrução possui custo 2, uma vez que o endereço da localização de memória [illegible] está na palavra que se segue à instrução.

3. A instrução [illegible] adiciona a constante 1 ao conteúdo do registrador [illegible]. Se possível, pode-se usar a constante 1 mais de uma vez aproveitando a próxima palavra que se segue à instrução.

4. A instrução [illegible] armazena o valor

conteúdo de conteúdo [illegible] + *conteúdo* [illegible] + *conteúdo de conteúdo* [illegible]

no destino [illegible]. O custo da instrução é três porque as constantes 4 e 12 são armazenadas nas duas próximas palavras que se seguem à instrução.

Algumas das dificuldades para se gerar código para esta máquina podem ser contornadas considerando-se um operador que gera um comentário de três endereços da forma [illegible], onde [illegible] são variáveis simples, constantes ou localizações de memória quaisquer, denotadas por seus nomes. Esse enunciado pode ser implementado para

---

[illegible]

vés de outras seqüências de diferentes de instruções. Aqui estão alguns exemplos:

1. [illegible] custo = 6

2. [illegible] custo = 6

Assumindo que [illegible] contenham os endereços de [illegible] respectivamente, podemos usar:

3. [illegible] custo = 2

Assumindo que [illegible] contenham os valores de [illegible] respectivamente, e que o valor de [illegible] não é necessário após a atribuição, podemos usar:

4. [illegible] custo = 1

Podemos ver que para gerar um bom código de boa qualidade para esta máquina precisamos saber, quais capacidades de endereçamento eficientemente. Existe um prêmio para se obter o valor de um endereço de memória em um registrador sempre possível, caso o mesmo venha a ser usado num futuro próximo.

9.3 GERENCIAMENTO DE MEMÓRIA EM TEMPO DE EXECUÇÃO

Como vimos no Capítulo 7, a semântica dos procedimentos numa linguagem pode determinar como os names são armazenados e como a instrução [illegible] execução. As informações necessárias durante a execução de um procedimento são mantidas num bloco de memória chamado de registro de ativação, armazenamento para os nomes locais ao procedimento também aparecem no registro de ativação.

Nesta seção discutimos que código gera para gerenciar os registros de ativação em tempo de execução. Na Seção 7.2 duas estratégias padrão de alocação de memória são apresentadas respectivamente: a alocação estática e a alocação de pilha. Na alocação estática, a posição de um registro de ativação na memória é fixada em tempo de compilação. Na alocação de pilha, um novo registro de ativação é empilhado para cada execução de um procedimento. O registro é desempilhado quando a ativação termina. Posteriormente, nesta seção, consideraremos o que o código para um procedimento pode ser feito para acessar os objetos de dados no registro de ativação.

Como vimos na Seção 7.2, um registro de ativação para um procedimento possui campos para armazenar parâmetros, resultados, informações sobre o estado da máquina, dados locais, temporários e outros itens. Nesta seção discutimos as estratégias de alocação usando um campo de estado da máquina para apontar o início de uma área de memória para os dados locais. Assumimos que os outros campos são manipulados como discutido no Capítulo 7.

Uma vez que a alocação e liberação de registros de ativação ocorrem como parte das seqüências de chamada e retorno, localizamos as seguintes cruciais de três endereços:

1. [illegible]
2. [illegible]
3. [illegible]
4. [illegible] um endereço de lugar para outros comandos

Por exemplo, a seqüência de três endereços para os procedimentos [illegible] Fig. 9.4, semelhantemente essas tipos de [illegible]. Um ponteiro [illegible] disposição de dados dos registros de ativação são comuns [illegible] gerador de código [illegible] informações [illegible] nos registros [illegible] esquemas [illegible] tabela de símbolos. Por uma questão de clareza [illegible] a disposição de dados na Fig. 9.4, ao invés da forma [illegible] entradas da tabela de símbolos.

Assumimos que a memória [illegible] tempo de execução seja dividida em áreas para código, dados estáticos e uma pilha. [illegible] na Seção 7.2 [illegible] adicionais. Para um [illegible] após [illegible] seção não é [illegible] aqui.

Alocação Estática

Consideremos o código necessário para implementar a [illegible] estática. Uma [illegible] código [illegible] interno [illegible] um [illegible] dado por uma sequência de duas instruções de máquina [illegible]. Uma instrução [illegible] que salva o endereço de retorno [illegible] e uma [illegible] que transfere o controle para o código [illegible] do procedimento. Portanto,

[illegible] para [illegible] chamada [illegible] contém
[illegible] código

Os atributos *staticsize* e *staticcode* [illegible] *staticsize* [illegible] constantes se referem, respectivamente, ao [illegible] registro de ativação e à primeira instrução do procedimento chamado. O código [illegible] [illegible] 20 na instrução [illegible] libera o registro de ativação [illegible] endereço de retorno [illegible] instrução [illegible] A partir das [illegible] são na Seção 9.2, as três [illegible] instruções na sequência de chamada [illegible] palavras [illegible].

O código para um procedimento termina com [illegible] retorno para o procedimento chamador. Exceto quando o primeiro procedimento [illegible] chamar outro, [illegible] sua instrução final é [illegible] para uma [illegible] presumivelmente retorna o controle [illegible] sistema operacional. [illegible] a partir do procedimento [illegible] [illegible]

[illegible] *staticsize* [illegible]

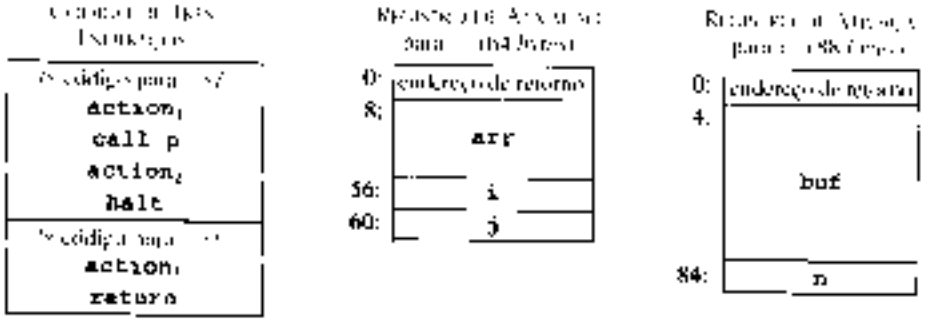


Fig. 9.4 [illegible] gerador de código

```
                                    ;; código para ...
100:                    ...
120:                    ...         ;; guarda o endereço de retorno 140
132:                    ...         ;; chamar ...
140:                    ...
140:                    ...

                                    ;; código para ...
200:                    ...
220:                    ...         ;; retornar para endereço salvo na
                                    localização 364

                                    ;; 300-363 abriga o registro de ativação para ...
300:                    ...         ;; endereço de retorno
304:                    ...         ;; dados locais para ...

                                    ;; 364-425 abriga o registro de ativação para ...
364:                    ...         ;; endereço de retorno
368:                    ...         ;; dados locais para ...
```

Fig. 9.5 Código-alvo para a execução da Fig. 9.4

que transfere o controle para o endereço 200 no início do registro de ativação.

Exemplo 9.1 O código na Fig. 9.5 é o código-alvo para duas das três procedures da Fig. 9.4. Usamos a pseudo-instrução ... para implementar o comando ... que representa um código de três endereços que não é relevante para esta discussão. Começamos alocando arbitrariamente o código para esses procedimentos nos endereços 100 e 200, respectivamente, e assumimos que cada instrução ... ocupe 20 bytes. Os registros de ativação para os procedimentos são alocados estaticamente, começando na localização 300 e 364, respectivamente.

As instruções que começam no endereço 100 implementam os três comandos

```
... ; ... ; ... ; ... ; ... ; ... ; ... ; ... ; ... ; ... ; ... ; ...
```

do primeiro procedimento. A execução, portanto, começa com a instrução ... no endereço 100. A instrução ... no endereço 120 salva o endereço de retorno 140 no campo de estado de máquina, que está à primeira palavra no registro de ativação de ... A instrução ... no endereço 132 transfere o controle para a primeira instrução do código-alvo do procedimento chamado.

Uma vez que 140 foi salvo no endereço 364 pela sequência de chamada acima, ... representa 140 quando o comando ... no endereço 220 for executado. O controle retorna, por conseguinte, para o endereço 140 e a execução do procedimento é retomada. ■

Alocação de Pilha

A alocação estática pode ser transformada em alocação de pilha pelo uso de endereços relativos no armazenamento de informações registros de ativação. A posição do registro para uma ativação de um procedimento não é conhecida até o tempo de execução. Na alocação de pilha, essa posição é normalmente armazenada em um registrador, de forma que as palavras no registro de ativação podem ser acessadas através de um deslocamento em relação a parte do valor no registrador. O modo de endereçamento indexado de nossa máquina-alvo é conveniente para esse propósito.

Os endereços relativos num registro de ativação podem ser considerados como deslocamentos contados a partir de qualquer posição conhecida no registro de ativação, como ilustra na Seção 7.3. Por razões de conveniência, usaremos deslocamentos positivos mantendo no registrador ... um apontador para o início do registro de ativação no topo da pilha. Quando uma sequência de procedimento procedimento chamador interrompe a transferência de controle para o procedimento chamado, o procedimento chamador incrementa ... liberando, por conseguinte, o registro de ativação do procedimento chamado.

O código para o primeiro procedimento inicializa a pilha através do estabelecimento de ... com um valor apontando para o início da área da pilha na memória

```
...  ...ializa_pilha,          ;inicializar a pilha
código para o primeiro procedimento
...                            ;término da execução
```

Uma sequência de chamada de procedimento inicializa a pilha através do endereço de retorno e transfere o controle para o procedimento chamado

```
...  ...chamador,...chamado_ele_próprio
...  ...#ret1  ;16 ...          ;salvar endereço de retorno
...  ...chamado,...ponto_de_entrada
```

O atributo *chamador.tamanho_do_registro* representa o tamanho de um registro de ativação, de forma que a instrução ... de soma ... aponta para o início do próximo registro de ativação. A constante #ret1 = 16 na instrução ... é o endereço da instrução que se segue a instrução de desvio ... e salvo no endereço apontado por ...

A sequência de retorno consiste em duas partes. O procedimento chamado transfere o controle para o endereço de retorno usando

```
...                            ;retornar ao chamador
```

a razão de ser usar ... na instrução de desvio é que precisamos de dois níveis de indireção. ... é o endereço da primeira palavra no registro de ativação e ... é o endereço de retorno salvo lá

Como é comum em seguida a procedimentos [illegible] para [illegible] e [illegible] [illegible].

A segunda parte da sequência de retorno está no chamador, que recomenda [illegible] para restaurar [illegible] após a substituição. [illegible] aponta para o início do registro de ativação do chamador.

[illegible]

Uma discussão mais ampla das sequências de chamada e dos bons códigos de [illegible] entre os procedimentos chamador e chamado figura na Seção 7.3.

Exemplo 9.2 O programa na Fig. 9.6 é uma condensação do código de três endereços para o programa Pascal discutido na Seção 2.1. O procedimento [illegible] recursivo, por conseguinte, mais de uma ativação do mesmo pode estar viva ao mesmo tempo.

Suponhamos que os tamanhos dos registros de ativação para os procedimentos [illegible] são conhecidos em tempo de compilação e são chamados *main*, *p* e *q*, respectivamente. A primeira palavra em cada registro de ativação é usada para o endereço de retorno. Assumiremos arbitrariamente que o código para esses procedimentos começa nos endereços 100, 200 e 300, respectivamente, e que a pilha começa em 600. O código alvo para o programa na Fig. 9.6 é como se segue:

```
[illegible]
código para main
action1
call q
action2
halt
código para p
action3
return
código para q
action4
call p
action5
call q
action6
call q
return
```

Fig. 9.6 Código de três endereços para [illegible] abreviado [illegible]

```
                                código para main
100:  [illegible]                 inicializar a pilha
108:  [illegible]
120:  [illegible]                 sequência de chamada inicial
136:  [illegible]                 empilha endereço de retorno
140:  [illegible]                 chamar q
152:  [illegible]                 restaurar fp
160:  [illegible]
180:  [illegible]
                                  código para p
200:  [illegible]
220:  [illegible]                 retornar
                                  código para q
300:  [illegible]                 desvio condicional para
320:  [illegible]
328:  [illegible]                 empilhar endereço de retorno
336:  [illegible]                 chamada para p
344:  [illegible]
352:  [illegible]
372:  [illegible]
380:  [illegible]                 empilhar endereço de retorno
388:  [illegible]                 chamar q
396:  [illegible]
404:  [illegible]
412:  [illegible]
432:  [illegible]                 empilhar endereço de retorno
440:  [illegible]                 chamada para p
448:  [illegible]
456:  [illegible]                 retornar

600                               a pilha começa aqui
```

Assumimos que [illegible] contém um desvio condicional para o endereço 456 da sequência de retorno de *p*, isto é, quando o procedimento recursivo *q* está terminando e chamar *p* novamente para sempre. No exemplo abaixo, consideramos a execução de um programa no qual a primeira chamada de *p* não ocorre imediatamente, mas todas as chamadas subsequentes a fazem.

Se *fp*, *sp* e *ss* são 20, 40 e 60, respectivamente, [illegible] é inicializado em 604, o início da pilha, pela primeira instrução no endereço 100. [illegible] está em 620, exatamente três deslocamentos de registro [illegible] para *p*, porque *locals* de *p* é 20. Subsequentemente, quando *p* chama *q*, a instrução no endereço 320 incrementa *sp* para 680, onde os campos de ligação de ativação para *q* começam. [illegible] será salva revertido para 620 após o controle retornar para *q*. Se as duas próximas chamadas de *q* retornarem imediatamente, o valor máximo de [illegible] durante esta execução será 680. Sobre-se enquanto aparelho colocação inicial na pilha em 739, uma vez que o registro de ativação para [illegible] começa em 680, se estende por 60 bytes [illegible]

Endereços para os Nomes, em Tempo de Execução

A estratégia de alocação de memória em tempo de execução e as disposições de dados no registro de ativação para um procedimento determinam como a memória para os nomes recebe acesso. No Capítulo 8, assumimos que um nome em uma instrução de três endereços é realmente um apontador para a entrada do nome na tabela de símbolos. Esse enfoque possui uma vantagem significativa, pois o compilador não precisa saber nada a mais sobre a localização em tempo de execução [illegible] cada nome se o compilador for migrado para uma nova máquina com uma diferente organização em tempo de execução é necessário mudar, por exemplo, o *display* pode ser mantido em registradores ao invés de na memória [illegible]. Por outro lado, a sequência específica de passos de acesso ao se gerar o código intermediário pode ser uma vantagem significativa [illegible] compilador otimizante, uma vez que permite ao compilador tirar vantagem de detalhes que seguem em vigência após a enumeração simples de três endereços.

Em qualquer caso, os nomes precisam eventualmente ser substituídos pelo código para se ter acesso [illegible] locações de memória. Consideraremos, por conseguinte, algumas alternativas de implementação sob o de três endereços de cópia [illegible]. Suponhamos que, após as declarações num procedimento terem sido processadas, a entrada da tabela de símbolos para *x* registre o endereço relativo 12 para *x*. Consideremos primeiro o caso em que *x* está numa área estaticamente alocada que começa no endereço *static*. Por conseguinte, o endereço efetivo de *x* em tempo de execução é *static* + 12. Apesar do compilador eventualmente determinar o valor de *static* + 12 em tempo de compilação, a posição da instrução para não ser conhecida quando o código intermediário para do acesso for inicialmente gerado. Nesse caso, faz sentido gerar um código de três endereços para "computar" *static* + 12, com a compreensão de que essa computação será colocada [illegible]

durante a geração de código, possivelmente pelo carregador, antes do programa rodar. A atribuição [illegible] se traduz em

[illegible]

Se a área estática começar no endereço 100, o código alvo para esse enunciado é

[illegible]

Por outro lado, suponhamos que nossa linguagem seja como Pascal e que um *display* seja usado para dar acesso aos pontos não locais, como discutido na Seção 7.4. Suponhamos, também, que o deslocamento separando do display registradores específicos localizados para cada ponto não local desejado do display estão pré-registrados. Podemos, então, traduzir [illegible] nos enunciados de três endereços

[illegible]

supondo [illegible] contém o endereço de [illegible]. Essa sequência pode ser implementada pela única instrução de máquina

[illegible]

Note-se que o valor no registrador [illegible] não pode ser determinado em tempo de compilação.

9.4 BLOCOS BÁSICOS E GRAFOS DE FLUXO

Uma representação de enunciados de três endereços, sob a forma de grafos, chamada de grafo de fluxo, é útil para a compreensão dos algoritmos de três endereços, mesmo que o grafo não seja explicitamente construído pelo algoritmo de geração de código. Os nós do grafo de fluxo representam computações e os lados representam o fluxo de controle. Na Capítulo 10 usamos extensivamente os grafos de fluxo para um programa como um todo para coletar informações a respeito do programa para intermediário. Alguns algoritmos de alocação de registradores usam grafos de fluxo para estimar os laços internos, nos quais um programa costuma passar a maior parte de seu tempo.

Blocos Básicos

Um *bloco básico* é uma sequência de enunciados consecutivos na qual o fluxo de controle entra no primeiro e deixa no último sem nenhuma parada ou possibilidade de ramificações, exceto ao final. A seguinte sequência de enunciados de três endereços forma um bloco básico:

[illegible] (9.1)

Um enunciado de três endereços [illegible] é dito *líder* se for o primeiro enunciado, ou [illegible] for o destino de um desvio condicional ou incondicional, ou [illegible] imediatamente após um desvio condicional. Dado um enunciado líder, podemos encontrar seu bloco básico incluindo o próprio líder e todos os enunciados até, mas não incluindo, o próximo líder ou o final do programa.

Algoritmo 9.1 Particionar uma sequência de enunciados em blocos básicos.

Entrada. Uma sequência de enunciados de três endereços.

Saída. Uma lista de blocos básicos com cada enunciado de três endereços exatamente em um bloco.

Método.

1. Determinamos primeiro o conjunto de líderes, os primeiros enunciados dos blocos básicos. As regras que usamos são as seguintes:
   i) O primeiro enunciado é um líder.
   ii) Qualquer enunciado que seja o objeto de um desvio condicional ou incondicional é um líder.
   iii) Qualquer enunciado que siga imediatamente um enunciado de desvio condicional ou incondicional é um líder.
2. Para cada líder, seu bloco básico consiste no líder e em todos os enunciados até, mas não incluindo, o próximo líder ou o final do programa.

Exemplo 9.3 Considere o fragmento de código-fonte mostrado na Fig. 9.7; ele multiplica o produto escalar de dois vetores [illegible] de comprimento 20. Uma lista de enunciados de três endereços que realiza essa computação em nossa máquina-alvo é mostrada na Fig. 9.8.

Vamos aplicar o Algoritmo 9.1 ao código de três endereços na Fig. 9.8 a fim de determinar seus blocos básicos. O enunciado (1) é um líder pela regra (i) e o enunciado (3) é um líder pela regra (ii), uma vez que o último enunciado pode saltar para o mesmo. Pela regra (iii), o enunciado seguinte a (12) também é um líder. Portanto, os enunciados (1) e (2) formam um bloco básico. O resto do programa, começando com o enunciado (3), forma um segundo bloco básico.

```
begin
    prod := 0;
    i := 1;
    do begin
        prod := prod + a[i] * b[i];
        i := i + 1
    end
    while i <= 20
end
```

Fig. 9.7 Programa para computar o produto escalar

Transformações sobre os Blocos Básicos

Um bloco básico computa um conjunto de expressões. Essas expressões são os valores dos nomes vivos na saída de um bloco. Dois blocos básicos são ditos *equivalentes* se computarem o mesmo conjunto de expressões.

Um número de transformações pode ser aplicado a um bloco básico sem mudar o conjunto de expressões computadas pelo mesmo. Muitas dessas transformações são úteis para melhorar a qualidade do código que será gerado, com base na análise a partir de um bloco básico. No próximo capítulo[1] mostraremos como um código global "bom" pode ser gerado a partir de transformações para reorganizar as computações nos blocos básicos, com o objetivo de reduzir as exigências globais de tempo e espaço para a execução do programa alvo final. Existem duas importantes classes de transformações locais que podem ser aplicadas aos blocos básicos, sem as transformações estruturais invariantes usadas [illegible]

[1] Examinaremos também [illegible] de [illegible] [illegible]

```
(1)   prod := 0
(2)   i := 1
(3)   t1 := 4 * i
(4)   t2 := a [ t1 ]
(5)   t3 := 4 * i
(6)   t4 := b [ t3 ]
(7)   t5 := t2 * t4
(8)   t6 := prod + t5
(9)   prod := t6
(10)  t7 := i + 1
(11)  i := t7
(12)  if i <= 20 goto (3)
```

Fig. 9.8 Código de três endereços para o produto escalar

Transformações Estrutura Invariantes

As transformações principais estruturais operam sobre os blocos básicos e incluem:

1. eliminação de subexpressões comuns
2. eliminação de código morto
3. renomeação de variáveis temporárias
4. intercâmbio de dois enunciados adjacentes independentes

Vamos agora examinar essas transformações em um pouco mais de detalhe. Novamente, assumimos que os blocos básicos não possuem aritméticos com apontadores ou chamadas de procedimentos.

1. *Eliminação de subexpressões comuns.* Considere um bloco básico

$$\begin{aligned} &\cdots \\ &\cdots \\ &\cdots \end{aligned} \tag{9.2}$$

Os segundo e quarto enunciados acima computam a mesma expressão, notadamente, $\cdots$, e portanto o bloco básico pode ser transformado no bloco equivalente

$$\begin{aligned} &\cdots \\ &\cdots \\ &\cdots \\ &\cdots \end{aligned} \tag{9.3}$$

Note-se que, apesar dos primeiros e terceiros enunciados em (9.2) e (9.3) parecerem ser a mesma expressão, o segundo enunciado pode redefinir $\cdots$. Consequentemente, o valor de $\cdots$ no terceiro enunciado é diferente do valor de $\cdots$ no primeiro, e o primeiro e terceiro enunciados não computam a mesma expressão.

2. *Eliminação de código morto.* Suponha que $\cdots$ é simplesmente usado, em ponto algum, após o enunciado $\cdots$. Esse enunciado, então, pode ser removido sem a computação ser mudada. O valor do bloco básico não é mudado.

3. *Renomeação de variáveis temporárias.* Suponhamos ter um enunciado da forma $\cdots$, onde $\cdots$ é uma temporária. Se mudarmos esse enunciado para $\cdots$, onde $\cdots$ é uma nova variável temporária, e mudarmos todos os usos desse instância de $\cdots$ para o valor desse bloco básico não é mudado. De fato, podemos sempre transformar um bloco básico num bloco equivalente no qual cada enunciado que define um temporário passa a definir um novo temporário. Chamamos a um tal bloco de *bloco básico em forma normal*.

4. *Intercâmbio de enunciados.* Suponhamos ter um bloco com dois enunciados adjacentes. Exemplo:

$$\begin{aligned} &\cdots \\ &\cdots \end{aligned}$$

Podemos, então, intercambiar os dois enunciados sem afetar o valor do bloco se e somente se nenhum $\cdots$ é igual a $\cdots$ e nenhum $\cdots$ é igual a $\cdots$. Note-se que um bloco básico em forma normal permite todas as trocas de enunciados possíveis.

Transformações Algébricas

Transformações algébricas incontroláveis podem ser usadas para mudar o conjunto de expressões computadas por um bloco básico em um conjunto algebricamente equivalente. As transformações úteis são aquelas que simplificam expressões ou substituem operações caras por outras mais baratas em conta. Por exemplo, um enunciado como

$$\cdots$$

ou

$$\cdots$$

pode ser chamados de um bloco básico sem modificar o conjunto de expressões que o bloco computa. O operador de exponenciação é normalmente

$$\cdots$$

assim muito requer uma chamada de função para ser implementado. Usando-se tais transformações algébricas, esse enunciado pode ser substituído pelo enunciado mais barato, porém equivalente

$$\cdots$$

As transformações algébricas são discutidas com mais detalhes na Seção 9.9 sobre otimização preditiva e na Seção 10.3 sobre substituição de blocos básicos.

Grafos de Fluxo

Podemos adicionar informações a respeito do fluxo de controle entre pares de blocos básicos que constituem um programa através da construção de um grafo dirigido chamado *grafo de fluxo*. Os nós dos grafos de fluxo são os blocos básicos. Um nó é distinguido como sendo o *início*. O bloco cujo líder é a primeira enunciado. Existe uma aresta dirigida do bloco B_1 para o bloco B_2 se B_2 pode ser seguir imediatamente a B_1 em alguma sequência de execução; isto é, se

1. existir um enunciado de desvio condicional ou incondicional da última enunciado de B_1 para a primeira enunciado de B_2, ou
2. B_2 seguir imediatamente a B_1 na ordem do programa e B_1 não terminar por um desvio incondicional.

Dizemos que B_1 é um *predecessor* de B_2 e que B_2 é um *sucessor* de B_1.

Exemplo 9.4 O grafo de fluxo do programa da Fig. 9.8 é mostrado na Fig. 9.9. B_1 é o nó início. Note-se que a última enunciado (o comando de desvio-if) foi substituído por um desvio equivalente para o início do bloco B_2. □

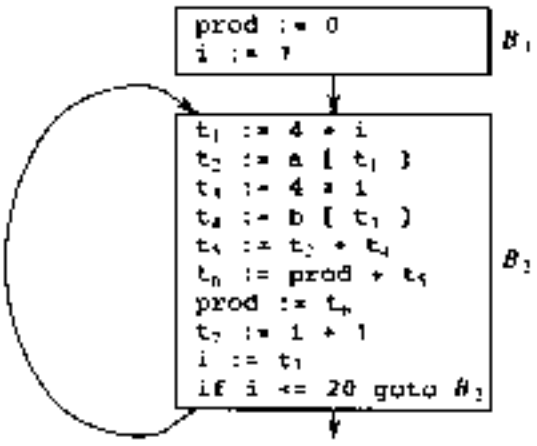


Fig. 9.9 Código de três endereços para o programa.

Representação dos Blocos Básicos

Os blocos básicos podem ser representados por uma variedade de estruturas de dados. Para exemplo, após particionar os enunciados de três endereços através do Algoritmo 9.1, cada bloco básico pode ser representado por uma referência que consiste em um ponteiro de início de quadruplo no bloco, seguido por um apontador para o líder imprimir no quadruplo do bloco e pelos listos de predecessores e sucessores do bloco. Uma alternativa é se fazer uma lista ligada das quadruplas em cada bloco. Referências explícitas aos enunciados de três endereços contidos dessa forma nos blocos básicos podem causar problemas se as quadruplas forem movidas durante a otimização de código. Por exemplo, se o bloco B_2 que abrange os enunciados de 4 a 12 no código intermediário da Fig. 9.9 fosse convertido para algum outro lugar de quatro quadruplas, esse enunciado [illegible] deveria ser modificado. Por conseguinte, preferimos fazer com que esses apontadores apontem para blocos em lugar de quadruplas, como fizemos na Fig. 9.9.

É importante notar que, num grafo de fluxo, um lado presente de um bloco B para um bloco B' não especifica as condições sob as quais o controle flui de B para B'. Isto é, o lado não nos diz se todo o conteúdo final de B ou existir um de seus enunciados finais, como a saída de B quando a condição for satisfeita ou quando não for. Essa informação pode ser recuperada quando necessário a partir do enunciado de desvio [illegible] B.

Laços

Num grafo de fluxo, o que é um laço é como se encontrar todos os laços? Na maioria das vezes é fácil responder a essa questão. Por exemplo, na Fig. 9.9 existe um laço que consiste no bloco B_2. As respostas para as duas questões, em geral, são um tanto sutis, e vamos examiná-las em detalhes no próximo capítulo. Para o presente, é suficiente dizer que um laço é uma coleção de nós num grafo de fluxo tal que

1. Todos os nós na coleção são *fortemente conectados*, isto é, a partir de qualquer nó no laço, em direção a qualquer outro, existe um percurso de comprimento um ou mais, inteiramente dentro do laço, e
2. A coleção de nós possui uma única *entrada*, isto é, um nó do laço, tal que a única forma de se atingir qualquer nó do laço a partir de um nó fora do mesmo consiste em se atingir primeiro o nó da entrada.

Um laço que não contém outros laços é, às vezes, chamado de um laço *interno*.

9.5 INFORMAÇÕES DE USO SUBSEQÜENTE

Nesta seção, coletamos informações de uso subseqüente (ou *next-use*) relativas aos blocos básicos. Se um nome associado a um registrador não é mais necessário, o registrador pode ser associado a algum outro nome. Isso decide se mantém um nome na memória ou não. Se o mesmo vier a ser subseqüentemente usado, pode ser aplicada as várias condições [illegible] usadas na Seção 8.6 para atribuir espaços aos valores de ativação. O gerador de código simples da mesma seção também aplica a atribuição de registradores. Como uma aplicação final, consideramos a atribuição de memória para nomes temporários.

Computando os Usos Subseqüentes

O uso de um nome em um enunciado de três endereços é definido como se segue. Suponhamos que o enunciado de três endereços i atribua um valor a x. Se o enunciado j tem x como um operando e se o controle pode fluir do enunciado i para j ao longo de um percurso que não tenha atribuições intervenientes a x, então dizemos que o enunciado j usa o valor de x computado em i.

Desejamos determinar, para cada enunciado de três endereços $x := y \; op \; z$, quais são os usos subseqüentes de x e y. Para o presente, não estamos preocupados com os usos fora do bloco básico que contém esse enunciado de três endereços, mas poderíamos se o desejarmos, tentar determinar se existe ou não um tal uso através da técnica de análise das variáveis vivas de (Capítulo 10).

Nosso algoritmo para determinar os usos subseqüentes realiza uma passagem para trás sobre cada bloco básico. Percorremos esquematicamente um bloco de enunciados de três endereços para encontrar as extremidades dos blocos básicos, e depois (Algoritmo 9.1) uma vez que os procedimentos podem ter alguns enunciados iniciais, os acrescentamos para a conclusão de cada bloco básico, e depois [illegible] de cada bloco básico.

Tendo encontrado o final de um bloco básico, examinamos cada enunciado de três endereços, começando com o último deles, para trás. Associamos a cada enunciado i com $x := y \; op \; z$ as informações sobre x e y (se x e y são nomes) e se eles possuem um uso subseqüente no bloco básico, caso não [illegible] se estão vivos ou não na saída do bloco. Se a análise de fluxo de dados discutida no Capítulo 10 tiver sido realizada, sabemos os nomes estão vivos na saída de cada bloco. Se nenhuma análise desse tipo tiver sido feita, podemos supor, [illegible] que todos os nomes não-temporários estão vivos na saída de cada bloco. Se os algoritmos que vamos considerar nesse [illegible] não precisam pesquisar que certos temporários sejam usados através dos blocos, os últimos também precisam ser considerados vivos. Seria uma boa idéia marcar quaisquer desses temporários, de forma [illegible] não termos de considerar todos os temporários como vivos.

Vamos supor que cada um dos enunciados de três endereços i [illegible] em nosso esquema tenha sido preenchido para trás. Fazemos então o seguinte:

1. Anexamos ao enunciado i as informações, correspondentes em cada uma das entradas da tabela de símbolos, relativas ao uso subseqüente e ao status de vida de x e y.
2. Na tabela de símbolos, fazemos x "não vivo" e sem uso subseqüente.
3. Na tabela de símbolos, fazemos y e z "vivos" e os usos subseqüentes de y e z [illegible] i. Note-se que a ordem de passos 2 e 3 não pode ser invertida, já que x pode ser y ou z.

Se o enunciado de três endereços i for da forma $x := op \; z$ ou $x := z$, os passos são os mesmos, ignorando y.

[illegible] T

[illegible] [illegible] [illegible] [

Memória para Nomes Temporários

Apesar de ser útil numa computação aritmética, em C um nome distinto é criado sempre que um nome temporário for necessário (veja o Capítulo 6). Para a posição relativa, o espaço precisa ser reservado para alocações desses temporários. O comprimento da área reservada para os temporários reaparece numa gestão geral de ativação (veja a Seção 7.2) através do uso de temporários.

Podemos, em geral, mapear os temporários para uma área de localização fixa, compacta, mas isso não é eficiente, pois simultaneamente muitos dos blocos básicos, as informações de uso subsequente poderão ser aplicadas para compactação dos dados. Para os temporários que são variáveis dentro dos blocos, o Capítulo 10 discute a análise de fluxo de dados necessária para computar a posição de vida.

Podemos reservar localizações de memória para temporários, uma de cada vez, à medida que cada um for encontrado. A primeira localização que não contenha um temporário vivo, naquele ponto do programa, é designada para o temporário recém-encontrado. Se um temporário não puder ser atribuído a nenhuma localização de memória previamente criada, adicionamos uma nova localização para a área de dados associada ao procedimento corrente. Em muitos casos, os temporários podem ser computados em registradores ao invés de localizações de memória, como na próxima seção.

Por exemplo, os seis temporários no bloco básico (6) podem ser compartilhar as mesmas localizações. Essas localizações correspondem a t_1 e t_2, como:

$$
\begin{aligned}
t_1 &:= t_3 * t_5 \\
t_2 &:= t_2 + t_4 \\
t_3 &:= t_2 \\
t_1 &:= t_3 * t_5 \\
t_2 &:= t_2 + t_4 \\
t_3 &:= t_2
\end{aligned}
$$

9.6 UM GERADOR DE CÓDIGO SIMPLES

A estratégia de geração de código desta seção produz um código-objeto para uma sequência de enunciados de três endereços. Considera um enunciado por vez, levando em se algum dos operandos está atualmente em registradores e tirando vantagem dessa fato, se possível. Por uma questão de simplicidade, assumimos que para cada operador binário existe uma instrução correspondente na linguagem-alvo. Assumimos igualmente que os resultados computados possam ser deixados em registradores até o final de cada bloco básico. A razão pela qual precisamos fazê-lo é que, após deixarmos um bloco básico, precisamos ter certeza de que as variáveis daquele bloco básico estão armazenadas em suas localizações de memória designadas, de modo que elas estejam disponíveis para uso em blocos básicos subsequentes. Em qualquer caso, não poderemos ser bem-sucedidos em assumir que um dado usado por um bloco básico aparece nos mesmos registradores, não importa como o controle tenha alcançado esse bloco. Descobrir uma forma de fazer o melhor uso possível dos registradores é muito importante. O algoritmo gerador de código simples armazena tudo quanto se deslocará através dos limites dos blocos básicos, bem como quando as chamadas de procedimentos são feitas. Posteriormente, consideramos como determinar algumas dessas coisas.

Podemos produzir um código-objeto para uma sucessão de três endereços $x := y \text{ op } z$ se gerarmos uma única instrução op com os mesmos, desde que y e z estejam em registradores. Depois disso, x será o único valor possível somente se os registradores y e z estiverem vivos e não contiverem quaisquer outros valores dos quais x também precisamos.

Se y e/ou z não estiverem numa localização de memória chamada y e/ou z, respectivamente, podemos gerar a sequência

$$\text{mov } y, R_y \qquad \text{custo} = 2$$

ou

$$\text{op } z, R_y \qquad \text{custo} = 2$$

onde R_y é um registrador após y ter sido armazenado e z não estiver em R_y. Após a sequência ser gerada, R_y contém x e x será armazenado, subsequentemente, se for necessário, na medida em que podemos obter seu valor a partir do registrador R_y. Existem muitas outras opções para considerar, dependendo de onde y e z estejam correntemente localizados e se y e/ou z estão subsequentemente vivos. Precisamos também considerar os casos em que x é um atributo de y ou z (ou ambos). Um número de casos que precisam ser posteriormente considerados aumenta se assumirmos que o operador op seja comutativo. Por conseguinte, podemos ver que a geração de código eficiente pode envolver um ampla número de casos e que esse deve provavelmente depender do contexto no qual um dado nó de três endereços é encontrado.

Descritores de Registradores e de Endereços

O algoritmo de geração de código usa descritores para controlar o conteúdo dos registradores e dos endereços para os nomes.

1. Um descritor de registradores controla o que está correntemente em cada registrador. É consultado sempre que um novo registrador for necessitado. Assumimos que inicialmente o descritor de registradores indique que todos os registradores estão vazios (se os registradores forem atribuídos através dos blocos, esse não será o caso). A regra que a geração de código progride é cada registrador irá abrigar o valor de zero ou mais nomes a um dado instante.

2. Um descritor de endereços controla a localização em localizações onde o valor corrente de um nome pode ser encontrado em tempo de execução. A localização poderia ser um registrador, uma localização da pilha, um endereço de memória ou algum conjunto desses elementos, porque uma vez copiado, o valor também permanece onde estava originalmente. Essas informações podem ser aumentadas com a tabela de símbolos e são usadas para determinar o método de acesso para um nome.

Um Algoritmo de Geração de Código

O algoritmo de geração de código toma como entrada uma sequência de enunciados de três endereços que estão em um bloco básico. Para cada enunciado $x := y \text{ op } z$ da forma $x := \text{op } y$ ou $x := y$, realizamos as seguintes ações.

1. Invocamos a função *obterreg* para determinar a localização L onde o resultado do computo $y \text{ op } z$ deverá ficar armazenado. L será idealmente sempre um registrador, mas poderia ser também uma localização de memória. Descreveremos *obterreg* logo a seguir.

2. Consultamos o descritor de endereços para y e determinamos y', a localização corrente de y. Uma das suas localizações é y.

Se o nome x não estiver em algum registrador disponível, e se x for um nome que não está vivo ao final do bloco básico, ou se não houver nenhum enunciado que o utilize, então o enunciado $x := y \text{ op } z$ poderá ser gerado como $\text{op } y, z$. Se y não estiver em um registrador, então uma cópia de y em x poderá ser gerada e considerada como uma opção de custo 2. Em todos os casos, x é marcado como sendo em seu próprio registrador, e não mais em y.

Se y e z estiverem em registradores, um enunciado $x := y \text{ op } z$ é gerado, e x é marcado como sendo em seu próprio registrador, mas não mais em y ou z. Se y ou z estiverem vivos e não estiverem em suas localizações de memória designadas, então uma cópia é gerada.

for y' para o registrador, caso o valor esteja tanto na memória quanto num registrador. Se o valor de y ainda não estiver em L, gerar a instrução MOV y', L a fim de colocar uma cópia de y em L.

3. Gerar a instrução OP z', L, onde z' é a localização corrente de z. De novo, preferir um registrador em vez de uma localização de memória, se z estiver em ambas. Atualizar o descritor de endereço de x para indicar que x está na localização L. Se L for um registrador, atualizar o seu descritor para indicar que o mesmo contém o valor de x e remover x de todos os demais descritores de registradores.

4. Se os valores correntes de y e/ou z não possuírem usos subseqüentes, não estão vivos na saída do bloco e estão em registradores, alterar, nesse caso, o descritor de registradores para indicar que, após a execução de x := y op z, aqueles registradores não irão mais conter y e/ou z, respectivamente.

Se o enunciado de três endereços corrente possuir um operador unário, os passos são análogos àqueles acima e, por conseguinte, omitimos os detalhes. Um caso especial importante é o enunciado de três endereços x := y. Se y estiver num registrador, mudar simplesmente os descritores de registradores e de endereços, de modo a que informem que o valor de x é agora encontrado somente no registrador que abriga o valor de y. Se y não tiver uso subseqüente e não estiver vivo na saída do bloco, o registrador já não abriga o valor de y.

Se y estiver somente na memória, poderemos, em princípio, registrar que o valor de x está na localização de y, mas essa opção iria complicar nosso algoritmo, uma vez que não poderíamos então mudar o valor de y sem salvaguardar o valor de x. Por conseguinte, se y estiver na memória, usaremos getreg para encontrar um registrador no qual carregar y e fazer do mesmo a localização de x.

Alternativamente, poderemos gerar uma instrução MOV y, x, o que seria preferível caso o valor de x não tenha uso subseqüente no bloco. É válido notar que a maioria das instruções de cópia, senão todas, será eliminada se usarmos o algoritmo de aprimoramento de blocos e propagação de cópias do Capítulo 10.

Uma vez que tenhamos processado todos os enunciados de três endereços num bloco básico, armazenamos, através de instruções MOV, aqueles nomes que estão vivos à saída e não estão em suas localizações de memória. Para realizar isso, usamos o descritor de registradores para determinar quais nomes foram deixados nos registradores, o descritor de endereços para determinar se o mesmo nome já não está em sua localização de memória e as informações sobre as variáveis vivas para determinar se um nome deve ou não ser armazenado. Se nenhuma informação sobre as variáveis vivas for computada pela análise de fluxo de dados entre os blocos, devemos assumir que todos os nomes definidos pelo usuário estão vivos ao final do bloco.

A Função *obtém_reg*

A função obtém_reg retorna a localização L para abrigar o valor de x para a atribuição x := y op z. Uma grande quantidade de esforços pode ser gasta na implementação desta função para produzir uma escolha perspicaz de L. Nesta seção, discutiremos um esquema simples, fácil de implementar, baseado nas informações de uso subseqüente coletadas na última seção.

1. Se o nome y estiver num registrador que não abrigue o valor de nenhum outro nome (relembremos que as instruções de cópia, tais como x := y, poderiam fazer com que um registrador guardasse o valor de duas ou mais variáveis simultaneamente) e y não estiver vivo, nem possuir uso subseqüente após a execução de x := y op z, devolver o registrador de y para L. Atualizar o descritor de endereço de y de forma a indicar que o mesmo já não está mais em L.

2. Em caso de falhar (1), devolver um registrador vazio para L, se existir um.

3. Caso (2) falhe, se x possuir um uso subseqüente no bloco ou se op for um operador que requeira um registrador, tal como de indexação, encontrar um registrador ocupado R. Armazenar o valor de R numa localização de memória (através de MOV R, M) se ainda não estiver na localização própria de memória M, atualizar o descritor de endereço de M e retornar R. Se R abrigar o valor de diversas variáveis, uma instrução MOV precisa ser gerada para cada variável que precise ser armazenada. Um registrador ocupado adequado poderia ser um cujo conteúdo seja referenciado mais distantemente no futuro ou um cujo valor esteja também na memória. Deixamos a escolha exata inespecificada, já que não existe um que tenha sido provado ser a melhor forma de se fazer a seleção.

4. Se x não é usado no bloco ou nenhum registrador ocupado adequado puder ser encontrado, selecionar L como a localização de memória de x.

Uma função obtém_reg mais sofisticada também consideraria os usos subseqüentes de x e a comutatividade de operadores ao determinar o registrador para abrigar o valor de x. Deixamos tais extensões de obtém_reg como exercícios.

Exemplo 9.5. A atribuição d := (a - b) + (a - c) + (a - c) poderia ser traduzida na seguinte seqüência de código de três endereços

t := a - b
u := a - c
v := t + u
d := v + u

Enunciados	Código Gerado	Descritor de Registradores	Descritor de Endereços
		registradores vazios	
t := a - b	MOV a, R0 SUB b, R0	R0 contém t	t em R0
u := a - c	MOV a, R1 SUB c, R1	R0 contém t R1 contém u	t em R0 u em R1
v := t + u	ADD R1, R0	R0 contém v R1 contém u	u em R1 v em R0
d := v + u	ADD R1, R0 MOV R0, d	R0 contém d	d em R0 d em R0 e memória

Fig. 9.10 Seqüência de código.

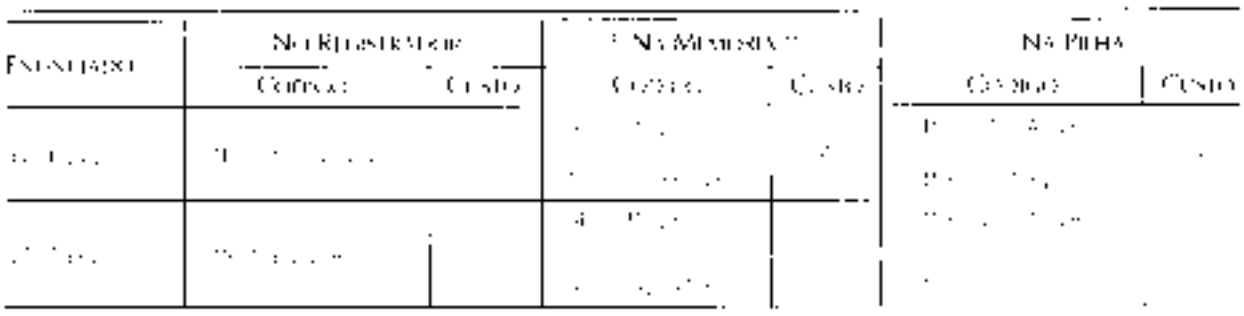


Fig. 9.11 Sequências de código para atribuições indexadas

com 1 byte ao fim. O algoritmo de geração de código dado acima produziria, para essa sequência de três endereços, a sequência de código mostrada na Fig. 9.10. Mostrados juntos estão os valores atuais dos descritores de registradores e de endereços, a medida que o processo de geração de código progride. Não mostrado no descritor de endereços é o fato de que x e t estão sempre na memória. Assumimos também que t é uma temporária, não estando em memória nenhuma quando o bloco básico explode e seus valores através de uma instrução.

A primeira chamada de *obter_reg* retorna $R1$ como a localização na qual se deve computar t. Uma vez que t não está em $R1$, gera-se uma instrução "t = $R1$". Atualizamos agora o descritor de registradores, indicando que $R1$ contém t.

A geração de código prossegue dessa maneira até que o último enunciado de três endereços $t = x - z$ tenha sido processado. Nesse ponto, x e t estão vivos, porque t não pode ser usado subsequentemente. Gera-se, então, $x = R1$ para armazenar o valor de x e o final do texto do bloco.

O custo do código gerado na Fig. 9.10 é 12. Poderíamos reduzir esse custo para 10 gerando $x = t$ imediatamente após a primeira subtração e removendo a instrução $t = x$. Outras heurísticas requerem um algoritmo de geração de código mais sofisticado. A razão para os ganhos está em que, em ocasiões corretas, a partir de um bloco básico de memória.

Gerando Código para Outros Tipos de Enunciados

As operações de indexação e de referenciamento nos enunciados de três endereços são tratadas da mesma maneira que as operações binárias. A tabela da Fig. 9.11 mostra as sequências de código geradas para os enunciados de atribuição indexada $x = y[i]$ e $x = *p$, assumindo que x está na memória e que y, i e p estão, cada um, em um registrador.

A localização corrente de $y[i]$ determina a sequência de código. Três possibilidades são tratadas, dependendo de i estar no registrador R_i, na localização de memória i ou na pilha, com deslocamento i em relação ao apontador para o registro de ativação de y estar no registrador R_y. O registrador R_y é aquele retornado quando a função *Obter_reg* é chamada para processar

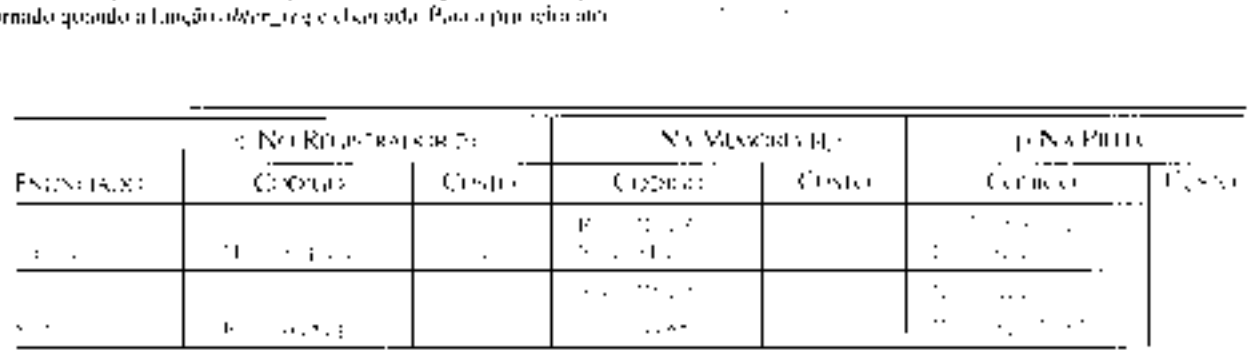


Fig. 9.12 Sequências de código para a atribuição de apontadores

Enunciados Condicionais

As máquinas implementam saltos condicionais em uma entre duas formas possíveis. Um deles testa o valor de um registrador em relação a uma das seguintes condições: negativo, zero, positivo, não negativo, diferente de zero ou não positivo. Uma tal instrução, que examina um endereço, tal como x, pode ser implementada por um salto condicional, seguindo-se uma instrução que carrega x em um registrador, e a seguir desviando se para x se o valor no registrador for negativo.

Em sequências curtas, comuns a muitas máquinas, há um conjunto de *códigos de condição*, de modo que a última operação executada em um registrador pode ser testada a negativo, zero ou positiva. Frequentemente, uma instrução de comparação explícita não é necessária, pois uma instrução que precisa de qualquer modo estabelecer o código de condição para positivo se x for maior do que zero. Uma instrução de máquina de desvio condicional realiza o desvio se o código de condição for um dos especificados, tais como "desvia para L se o código de condição for negativo ou zero". Para exemplificar, a instrução $x = y - z$ poderia ser implementada através de

Se estamos gerando código para uma máquina com códigos de condição, e um operando é um descritor do código de condição na memória em que geramos o código. Esse descritor informa a maneira que estabeleceu o código de condição pela última vez, ou seja, de modo que, se o código de condição foi estabelecido dessa forma pela última vez. Por conseguinte, poderíamos implementar

[illegible]

através de

[illegible]

se o descritor informar que o código de condição foi determinado [illegible] valor de [illegible].

9.7 ALOCAÇÃO E ATRIBUIÇÃO DE REGISTRADORES

As instruções envolvendo somente operandos do tipo registrador são mais curtas e rápidas do que aquelas que envolvem operandos na memória. Por conseguinte, a utilização eficiente dos registradores é importante na geração de um código de boa qualidade. Esta seção apresenta várias estratégias para decidir que valores num programa devem residir em registradores, alocação de registradores e para designar cada valor devem residir, atribuição de registradores.

Uma abordagem para a alocação e atribuição de registradores é associar valores específicos num programa a objetos específicos registradores. Por exemplo, uma decisão pode ser tomada para alocar endereços base a um grupo de registradores, e o compilador pode reservar, digamos, pilha em tempo de execução a um registrador fixo, o ponteiro.

Esse enfoque possui a vantagem de simplificar o projeto do compilador. Sua desvantagem está em que, aplicado excessivamente, ao pé da letra, utiliza os registradores inefficientemente: certos registradores podem ser usados somente em uma ou duas sub-sequências de código, enquanto que são penalizadas cargas e armazenamentos desnecessários de outros. Apesar de tudo, é razoável na maioria dos ambientes de computação se reservar uns poucos registradores como base, apontadores de pilha e de bibliotecas a fim de permitir que os registradores remanescentes possam ser usados pelo compilador, na medida em que sejam valores suficientes.

Alocação Global de Registradores

O algoritmo de geração de código na Seção 9.6 usou os registradores para guardar valores pelo decorrer de um único bloco básico. No entanto, todas as variáveis vivas foram armazenadas no final de cada bloco. Para eliminar alguns desses armazenamentos e cargas correspondentes, poderíamos projetar para associar registradores às variáveis mais frequentemente usadas e manter esses registradores consistentes ao longo dos limites dos blocos (*sub-rotinas*). Uma vez que os programas gastam a maior parte de tempo nos seus laços internos, uma abordagem natural para a atribuição global de registradores é tentar manter um valor frequentemente usado num registrador livre ao longo de um laço. Para o momento, vamos assumir que conhecemos a estrutura de laço de um grafo de fluxo e que sabemos onde os valores são computados num bloco básico e são usados fora dessa. O próximo capítulo cobre as técnicas para computar essas informações.

Uma estratégia para a alocação global de registradores é atribuir algum número fixo de registradores para guardar a maior parte dos valores ativos em cada laço mais interno. Os valores selecionados pelos vários laços diferentes em diferentes [illegible] registradores atribuídos podem ser usados para guardar os valores vivos ao redor do laço, como na Seção 9.6. Esse enfoque possui a desvantagem de que um número fixo de registradores não é sempre o número correto a ser disponível para a alocação global de registradores. Ainda assim, a penalidade é pequena de implementar e foi usada em Fortran H, o compilador otimizante para a série de máquinas IBM/360 [Lowry e Medlock [1969]].

Em linguagens como C e Bliss, o programador pode realizar diretamente a alocação de registradores através de declarações de forma a manter certos valores em registradores pela duração de um procedimento. O compilador respeita essas declarações de registradores, pode ainda otimizar programas nos quais o programador não declarou explicitamente o perfil de comportamento do programa.

Contadores de Utilização

Um método simples para se determinar as economias a serem realizadas ao atender-se uma variável *v* num registrador pela duração de um laço *L* é contabilizar que cada uso de *v* dentro de *L* economiza uma unidade de custo para cada referência a *v*, se *v* estiver em registrador. Entretanto, se usarmos o enfoque de seção anterior para gerar o código de um bloco, existe uma boa chance de que *v* apareça somente uma vez no bloco, ou seja a permanecer num registrador, se existirem outros ainda suspeitos de naquele bloco. Por conseguinte, contabilizamos com uma economia de um para cada uso de *v* no laço *L* que não seja precedido por uma atribuição a *v* no seu bloco. Também contabilizamos duas unidades se pudermos evitar um armazenamento de *v* no final de um bloco. Por conseguinte, se *v* for alocado a um registrador, contamos com um ganho de dois para cada bloco de *L* para o qual *v* está viva à saída e no qual é atribuído um valor.

No lado do débito, se *v* está viva à entrada do cabeçalho do laço, precisamos carregar *v* em seu registrador exatamente antes de entrar no laço *L*. Assumimos outras coisas. Se *v* é alterado no bloco de saída *B* de laço *L*, ou que *v* esteja viva à entrada de algum sucessor de *B* fora de *L*, precisamos armazenar *v* na saída de *B*. Entretanto, se suponhamos que um laço seja repetido muitas vezes, podemos negligenciar esses débitos uma vez que ocorrem somente uma vez a cada vez que entramos no laço. Por conseguinte, uma estimativa aproximada para o lucro a ser realizado pela alocação de um registrador *v* dentro de um laço *L* é

$$\sum_{\text{blocos } B \text{ em } L} \text{Utilização de } v \text{ em } B + 2 \times \text{Vivo em } v, B \qquad (9.4)$$

onde *utilização de v em B* é o número de vezes que *v* é usado em *B* antes de qualquer definição de *v* dentro de *B* e *Vivo em v, B* é 1 se *v* está viva à saída de *B* e recebe uma atribuição de valor em *B* e 0 caso contrário. Note se que (9.4) é aproximada porque nem todos os blocos num laço são executados com igual frequência, e também porque está baseada na suposição de que um laço é executado muitas vezes. Em outras palavras, terá que ser desenvolvida uma fórmula análoga e possivelmente diferente de (9.4).

Exemplo 9.6 Considere todos os blocos básicos do laço mais interno de nosso exemplo na Fig. 9.13, todos os enumerados de dentro do ciclo original e incondicional. Suponha também que os registradores [illegible] são alocados para guardar valores ao longo do laço. Por uma questão de conveniência, as variáveis vivas à entrada e à saída de cada bloco são listadas na Fig. 9.14, juntamente com as alterações de cada bloco, respectivamente. Existem alguns pontos sutis a respeito das variáveis vivas, as quais endereçamos no Capítulo 10. Por exemplo, notamos que ambos, *c* e *i*, estão vivos no final de *B*, mas dessas, somente *i* está viva à entrada de *B*, e somente *c* à entrada de *b*. Em geral, as variáveis vivas ao fim de um bloco são a união daquelas vivas ao início de cada um de seus blocos sucessores.

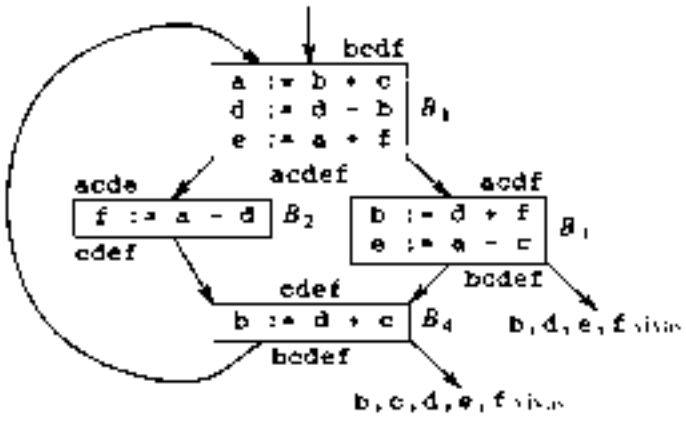


Fig. 9.13 Grafo de fluxo para um bloco progressão aritmética.

Além de avaliar (9.4) para $i = 1$, observamos que o resultado de B_1 é recebido em valor no registrador usado na saída de B_2 e B_3. Por conseguinte,

$$\sum_{i=1} \text{custo}(i, B_1) = 2$$

Igualmente, $\text{custo}(i, B_2) = 0$, uma vez que i é definida em B_2 antes de ser, quer em seguida, usada. $\text{custo}(i, B_3) = 1$ e $\text{custo}(i, B_4) = 0$. Por conseguinte,

$$\sum_{i=1} \text{custo}(i, B) = 3$$

Essa forma o valor de (9.4) para $i = 1$. Esse é quanto custaria de custo pode ser economizado selecionando-se i para um dos registradores globais. Os valores de (9.4) para $i = 2$, e serão 3, 6 e 3, respectivamente. Por conseguinte, devemos selecionar i para um registrador e i para o respectivamente. Usar para economizar em lugar de setia uma contribuição custo ao benefício aparente. A Fig. 9.14 mostra o código de montagem gerado a partir de Fig. 9.13, assumindo que a estratégia de Seção 9.6 é usada para gerar código para cada bloco. Não mostramos o código gerado para os devidos condicionais e incondicionais entre blocos, uma vez que ele é o mesmo da Fig. 9.13, e, por conseguinte, não influencia o código gerado como um todo. Lembre-se, contudo, que se não adotássemos a estratégia de reservar R_1 e R_2 para i, teríamos de usar

$$\begin{array}{l} \\ \\ \end{array}$$

para B_1, exemplificando uma atribuição uma vez que i não está viva à saída do bloco B_4. Um ganho similar poderia ser obtido em B_1, B_2, B_3 e B_4.

Atribuição de Registradores para Laços Mais Externos

Tendo atribuído os registradores e gerado código para os laços mais internos, podemos aplicar a mesma estratégia para laços progressivamente mais externos. Se um laço mais externo L_0 possui um laço mais interno L_1 que já foi alocado, que fazemos nesses registradores em L_0 mais provável que tenha registradores alocados em L_0. No entanto, se um nome tem um registrador alocado no laço L_1 mas não em L_0, precisamos armazená-lo à saída para L_1 e carregá-lo quando ele entra em um bloco de L_0 e L_1. Semelhantemente, se escolhemos alocar para um registrador em L_0 mas não em L_1, precisamos carregá-lo à entrada para L_1 e armazená-lo à saída. Deixamos como um exercício a tarefa de um critério para selecionar nomes a terem registradores alocados mais especificamente em L_0 (daqueles que não estão já alocados para todos os laços aninhados dentro de L).

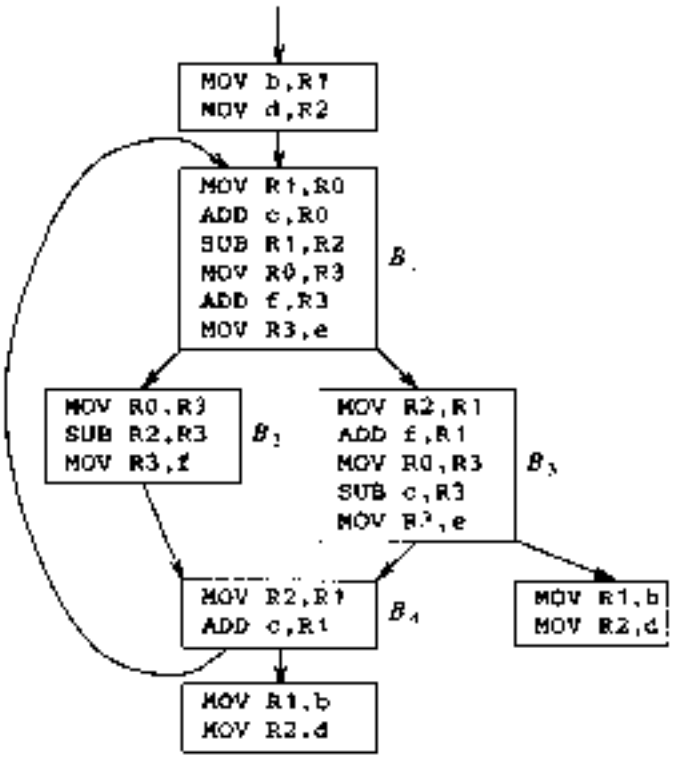


Fig. 9.14 Sequência de código ilustrando a uso global de registradores.

Alocação de Registradores Colorindo-se um Grafo

Quando um registrador é necessário para uma computação mas todos os registradores disponíveis estão em uso, o conteúdo de um dos registradores precisa ser armazenado (ou *derramado*) numa localização de memória, a fim de liberar um registrador. A coloração de grafos é uma técnica sistemática sofisticada para alocar registradores e manipular derramamentos.

Neste método, são usadas duas passagens. Na primeira, as instruções da linguagem alvo são selecionadas como se houvesse um número ilimitado de registradores simbólicos, exceto que os nomes usados no código intermediário se tornam os nomes de registradores simbólicos. Se o acesso às variáveis requer instruções especiais, tais como instruções de pilha ou de *display*, suponhamos também que quantidades que são só acessadas escalares que as mesmas sejam guardadas em registradores, exceto para cada ocorrência. Normalmente, esses nomes são diretamente traduzidos para modos de acesso para o endereço mais curto de uma instrução de máquina. Se o acesso é mais complexo, as sub-árvore deve ser calculada em várias instruções e armazenada em um registrador temporário (ver Cap. 8) pode necessitar ser criado.

Uma vez que as instruções estejam sido selecionadas, na segunda passagem atribuímos registradores físicos aos símbolicos. Aqui encontramos uma atribuição que minimize o custo das derramagens.

Na segunda passagem, é construído um *grafo de interferência* de registradores para cada procedimento. Os nós deste grafo são registradores simbólicos e cada arco conecta dois nós que devem ser mapeados

```
(1)   t1 := 4 * i
(2)   t2 := a [ t1 ]
(3)   t3 := 4 * i
(4)   t4 := b [ t3 ]
(5)   t5 := t2 * t4
(6)   t6 := prod + t5
(7)   prod := t6
(8)   t7 := i + 1
(9)   i := t7
(10)  if i <= 20 goto (1)
```

Fig. 9.15 Código de três endereços para o Exemplo 9.6

a partir das folhas representadas abaixo. As folhas 6 presentes nos valores iniciais dos nomes que subscrevemos com 0 para evitar confusão com os valores de variáveis após a execução de instruções. Os nós interiores também são rotulados por símbolos operadores, exceto aqueles que possuem um ou mais rótulos adicionais. Neste caso, os rótulos adicionais são listados depois do operador.

2) Os nós internos são rotulados por um símbolo de operador.

3) Aos nós também podem ser atribuídos uma ou mais variáveis de entrada. A intenção é que os nós internos representem valores computados nos derradeiros pontos que alcançam um nó sejam considerados iguais após o bloco.

É importante que se continue os GDAs com os grafos de fluxo. Cada nó de um grafo de fluxo pode ser representado por um GDA, uma vez que cada nó do grafo de fluxo está em lugar de um bloco básico.

Exemplo 9.7. A Fig. 9.16 apresenta o GDA correspondente aos três endereços correspondentes ao bloco B_3 da Fig. 9.9. Os números de enumerado começando de 1. Folhas são rotuladas por uma variável de entrada. O GDA é construído de acordo com o algoritmo da Fig. 9.16. Descreveremos sua aplicação ao GDA após termos descrito o algoritmo. Para o momento, vamos observar que cada nó do GDA representa uma fórmula em termos das folhas. Esses valores são passados pelas variáveis e constantes que calculam no bloco. Por exemplo, o nó rotulado * na Fig. 9.16 representa a fórmula

$$\ldots * \ldots$$

onde o valor da primeira ocorrência de i é o valor desse i no início do bloco, e a segunda ocorrência de i é o valor de i depois da atribuição $i := i + 1$.

Construção de GDAs

Para se construir um GDA para um bloco básico, processamos um comando de cada vez. Quando encontramos uma instrução de forma $x := y \; op \; z$, primeiramente procuramos pelos nós que representam os valores correntes de y e z. Esses poderão ser folhas ou poderão ser nós que

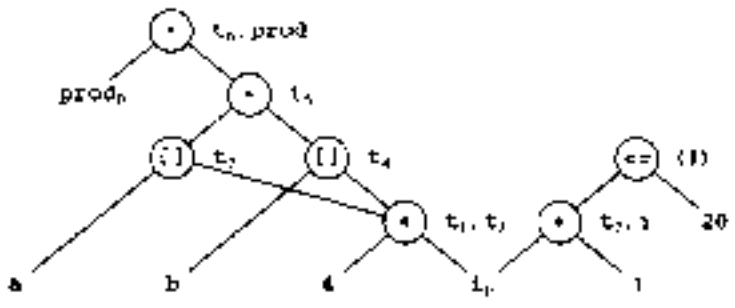


Fig. 9.16 GDA para o bloco da Fig. 9.15

[illegible] do GDA, se [illegible] processamento dos atributos sobre os enumerados anteriores do bloco. Criamos um vértice rotulado [illegible] e ligamos os nós correspondentes aos filhos esquerda e direita do nó para [illegible]. Rotulamos cada vértice com [illegible]. Novamente, se precisarmos [illegible] não é necessário que [illegible] adicionamos o vértice nó do GDA, mas sim através [illegible] do nó existente o rótulo adicional [illegible].

Dois detalhes devem ser mencionados. Primeiro, se [illegible] previamente rotulado algum vértice, removemos o rótulo anterior, uma vez que cada vértice [illegible] rotulado. Segundo, para uma atribuição da forma [illegible], nada fazemos. Ao invés, atualizamos o rótulo [illegible] da lista de nomes para [illegible] corrente" [illegible].

Fornecemos agora a rotina para computar o GDA a partir de um bloco. O algoritmo é quase o mesmo que o Algoritmo 9.1, exceto pela lista adicional de identificadores que anotamos a cada nó. O leitor deve estar ciente de que esse algoritmo pode não operar corretamente se existirem atribuições para [illegible]; se existirem atribuições [illegible] de aponta [illegible] pode ser determinado através de duas formas nomes, devido a comandos [illegible] ou a [illegible] parâmetros [illegible] e usuais de linguagens [illegible] procedimentos. Discutimos as modificações necessárias para [illegible] situações ao final desta seção.

Algoritmo 9.2. Construção de um GDA.

Entrada. Um bloco básico.

Saída. Um GDA para o bloco básico contendo as seguintes informações:

1. Um vértice para cada [illegible]. Para as folhas, o rótulo é um identificador ou constante são permitidas e para os nós internos, um símbolo [illegible].

2. Para cada nó uma lista [illegible] contendo zero ou mais identificadores anotados [illegible] constantes não são permitidas aqui.

Método. Assumimos que as estruturas de dados apropriadas estejam disponíveis para criar nós com um ou dois filhos, com a distinção entre filho à "esquerda" e filho à "direita" no último caso. Também dispomos de uma estrutura [illegible] para o rótulo para cada nó e facilidade de criar uma lista [illegible] de identificadores anotados a cada nó.

Admitimos também que esses componentes previamente [illegible] contêm ponteiros [illegible] todos os identificadores [illegible] constantes para os quais [illegible] nós. Cada ponteiro se refere [illegible] rotulada por um identificador [illegible] que pode ser [illegible] ou [illegible] identificador [illegible] ou um símbolo [illegible] constante. Assumimos a existência da função *novo(identificador)*, que [illegible] que constrói nós no GDA, retorna um nó recém-criado [illegible] rotulado com *identificador*. Internamente, *novo* [illegible] do GDA que [illegible], e cada vez que aquele identificador [illegible] processar [illegible] construção do GDA. Na prática, [illegible] *novo* [illegible] *identificador* indicaria o valor de *nó(identificador)*.

O processo de construção do GDA é [illegible] passos, de 1 a 3, para cada enumerado da [illegible], sucessivamente. De início, assumimos que não existem nós que [illegible] para todos os argumentos. Suponhamos que o enumerado "corrente" de três endereços seja [illegible]. Referimo-nos a esses como casos [illegible]. Tratamos um operador relacional [illegible] como [illegible] semelhantes [illegible]

1. Se [illegible] foi incluído [illegible] folha rotulada [illegible] que [illegible] igual a esse nó. No caso [illegible], [illegible] rotulada [illegible] e fazer [illegible] folha [illegible].

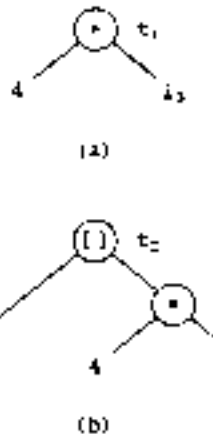


Fig. 9.17 Passos no processo de construção do GDA

2. No caso [illegible], determinar se existe um nó rotulado [illegible] cujos filhos [illegible] esquerda seja [illegible] e cujo filho à direita seja [illegible]. Essa verificação é para capturar as subexpressões comuns. Caso [illegible] não [illegible] qualquer dos casos seguintes, [illegible] nó [illegible] do [illegible]. No caso [illegible], determinar se existe um nó rotulado [illegible] cujo único filho seja [illegible]. Se não existir, criar um [illegible] e seja [illegible] nó recém-criado. No caso [illegible] seja [illegible].

3. Remover [illegible] da lista de identificadores anotados para [illegible] incluir [illegible] à lista anotada de identificadores para o nó encontrado em 2 ou fazer [illegible] igual a [illegible].

Exemplo 9.8. Vamos rotular os blocos da Fig. 9.15 e ver como o GDA da Fig. 9.16 é construído a partir deles. O primeiro enumerado é [illegible]. Após o passo 1, criamos [illegible] folhas rotuladas [illegible], como mostra a subárvore (a) a seguir. Para simplificar a distinção dos rótulos dos identificadores anotados nas figuras, mas subscrito vazio realmente faz parte dos rótulos. No passo 2, criamos um nó rotulado [illegible] no passo 3, anotamos a identificador [illegible] mesmo. A Fig. 9.17(a) mostra o GDA nesse estágio.

Para o segundo enumerado, [illegible], criamos uma nova folha rotulada por [illegible] encontramos o nó previamente criado, [illegible]. Criamos, também, um nó rotulado [illegible] pelo qual anotamos os nós para [illegible] como filhos.

Para o enumerado [illegible], [illegible] determinamos que [illegible] existe. Como o operador [illegible] não [illegible] para o enumerado [illegible], mas, ao invés, adicionamos [illegible] à lista de identificadores para o nó [illegible]. O GDA resultante é mostrado na Fig. 9.17(b). O método dos números de valor da Seção 5.2 pode ser usado para descobrir rapidamente que [illegible] existe.

Concluímos o leitor a completar a construção do GDA. Mencionamos sugestões passos dados para o enumerado [illegible]. Antes de enumerado [illegible], [illegible] a folha rotulada [illegible]. O enumerado [illegible] é uma instância [illegible], por conseguinte obtemos [illegible] anotamos [illegible] final da sua lista de identificadores, e atribuímos [illegible]. Esse é um dos comandos [illegible], então, após [illegible] — sendo o valor de [illegible] usado para um identificador. É a ausência [illegible] que assegura que um nó com [illegible] filho à esquerda do nó de operador [illegible] usado para o enumerado [illegible].

Aplicações de GDAs

Existem diversos itens de informação que podemos obter a partir do que está contido [illegible] Algoritmo 9.2. Primeiro, observamos que detectamos automaticamente as subexpressões comuns. Segundo, podemos determinar [illegible] que identificadores possuem seus valores usados

[illegible] [illegible] [illegible] [

[illegible] do bloco são exatamente aqueles para os quais uma folha é criada no passo (1) em algum instante. Em seguida, podemos determinar que enunciados computam valores que poderiam ser usados fora do bloco. São exatamente aqueles enunciados S cujos nós n construídos ou encontrados no passo (2) ainda possuem [illegible] ao final da construção do GDA, onde [illegible] é o identificador atribuído pelo enunciado S. (Equivalentemente [illegible] ainda é um identificador de [illegible].)

Exemplo 9.9. No exemplo 9.8, todos os enunciados atendem à restrição acima porque as únicas vezes em que [illegible] é redefinido [illegible] para [illegible] o valor anterior de [illegible] deixa de ser uma folha. Por conseguinte, todos os nós interiores podem ter seus valores usados fora do bloco. Vamos supor agora que tivéssemos inserido antes do enunciado (9) um novo enunciado [illegible] que atribuísse um valor a [illegible]. No enunciado [illegible] criaríamos um nó [illegible] (com [illegible]). No entanto, no enunciado (9) redefiniríamos [illegible]. Consequentemente, o valor computado no enunciado [illegible] não poderia ser usado fora do bloco. □

Um outro uso importante no qual o GDA pode ser colocado é o de reconstruir uma lista simplificada de quádruplas, tirando-se partido das subexpressões comuns e não realizando-se as atribuições da forma [illegible] a menos que sejam absolutamente necessárias. Isto é, sempre que um nó possuir mais de um identificador em sua lista, verificamos quais daqueles identificadores, se algum, são necessários fora do bloco. Como mencionamos, encontrar as variáveis vivas ao final de um bloco requer uma análise de fluxo de dados chamada de "análise das variáveis vivas", discutida no Capítulo 10. Entretanto, em muitos casos podemos assumir que nenhum nome temporário, tal como [illegible] na Fig. 9.15, seja necessário fora do bloco (mas fique-se atento sobre como as expressões lógicas são traduzidas; uma expressão pode se espalhar por sobre diversos blocos básicos).

Podemos, em geral, avaliar os nós interiores de um GDA em qualquer ordem que seja uma classificação topológica do mesmo. Numa classificação topológica, um nó não é avaliado até que todos os seus filhos que sejam nós interiores tenham sido avaliados. À medida que avaliamos um nó, atribuímos seu valor a um dos identificadores atrelados [illegible] preferindo-se aquele cujo valor seja necessário fora do bloco. Não podemos, entretanto, escolher [illegible] se existir um outro nó [illegible] cujo valor foi também guardado por [illegible] tal que [illegible] tenha sido avaliado e esteja ainda "vivo". Aqui, definimos [illegible] como vivo se seu valor for necessário fora do bloco ou se [illegible] possuir um pai que não tenha sido ainda avaliado.

Se existirem identificadores adicionais atrelados [illegible] cujos valores sejam também necessários fora do bloco, atribuímos aos mesmos através dos enunciados [illegible]. Se [illegible] não possuir de todo identificadores atrelados (isto poderia acontecer se, digamos, [illegible] fosse sido criado por uma atribuição [illegible] mas [illegible] fosse recebido subsequentemente uma reatribuição), criaríamos um novo nome temporário para guardar o valor de [illegible]. O leitor deve estar consciente de que na presença de uma atribuição de *array* ou de apontador, nem toda classificação topológica de um GDA é permitida; proveremos endereçar tal matéria em breve.

Exemplo 9.10. Vamos reconstruir um bloco básico da Fig. 9.16, ordenando os nós na mesma ordem em que foram criados [illegible]. Note-se que os enunciados (3) e (7) do bloco original não criaram novos nós, mas adicionaram os rótulos [illegible] e [illegible] às listas dos nós [illegible] respectivamente. Assumimos que nenhum dos temporários [illegible] seja necessário fora do bloco.

Começamos com o nó que representa [illegible]. Este nó possui dois identificadores atrelados, [illegible] e [illegible]. Vamos pinçar [illegible] para guardar o valor [illegible] de forma que o primeiro enunciado reconstruído é

$$[illegible]$$

exatamente como no bloco original. O segundo nó considerado é rotulado [illegible]. O enunciado construído para esse nó é

$$[illegible]$$

como antes. O nó considerado em seguida é rotulado [illegible] a partir do qual geramos o enunciado

$$[illegible]$$

O último enunciado usa [illegible] como um argumento ao invés de [illegible] como no bloco básico original, porque [illegible] é o nome escolhido para carregar o valor de [illegible].

Em seguida, consideramos o nó rotulado [illegible] e geramos o enunciado

$$[illegible]$$

Para o nó rotulado [illegible] selecionamos [illegible] para carregar o valor, uma vez que esse identificador e não [illegible] irá presumivelmente ser necessitado fora do bloco. Como [illegible] o temporário [illegible] desaparece. O próximo enunciado gerado é

$$[illegible]$$

Similarmente, escolhemos [illegible] ao invés de [illegible] para carregar o valor de [illegible]. Os dois últimos enunciados gerados são

$$[illegible]$$

Note-se que os dez enunciados da Fig. 9.15 foram reduzidos a sete tirando-se partido das subexpressões comuns expostas durante o processo de construção do GDA e eliminando-se as atribuições desnecessárias. □

Arrays, Apontadores e Chamadas de Procedimentos

Consideremos o bloco básico:

$$[illegible] \qquad (9.5)$$

Se usarmos o Algoritmo 9.2 para construir o GDA para (9.5), [illegible] se tornaria uma subexpressão comum e o bloco "otimizado" se tornaria

$$[illegible] \qquad (9.6)$$

Entretanto, (9.5) e (9.6) computam valores diferentes para [illegible] no caso de [illegible]. O problema é que quando atribuímos a um *array* [illegible] podemos estar modificando o valor r da expressão [illegible] mesmo que [illegible] não mudem. Por conseguinte, é necessário que, ao se processar uma atribuição a um *array* [illegible] matemos todos os nós rotulados [illegible] cujos argumentos à esquerda sejam mais ou menos uma constante (possivelmente zero)[1]. Isto é, tornamos esses nós inelegíveis para receber um rótulo de identificador adicional, impedindo-os de serem falsamente reconhecidos como subexpressões comuns. É dessa forma requerido que mantenhamos um *bit* para cada nó, informando se o mesmo foi ou não morto. Sobretudo, para cada *array* [illegible] mencionado no bloco é conveniente ter uma lista de todos os nós correntemente não mortos mas que terão de sê-lo caso atribuamos a um elemento de [illegible].

[1] Note-se que o argumento [illegible] indicando o nome do *array* poderia ser o próprio [illegible] ou uma expressão como [illegible]. No último caso, o [illegible] seria o nó rotulado [illegible] com filho [illegible].

Um problema similar ocorre se temos uma atribuição da forma [illegible] em que [illegible] é um ponteiro. Se essa sequência for traduzida para código de máquina exatamente no GDA acima, o endereço apontado por [illegible] será sempre o mesmo endereço de memória. Se o valor de [illegible] for modificado por uma atribuição subsequente a [illegible], precisaremos criar uma nova célula para [illegible] e associar esse valor em lugar de 0. Obviamente, todas as instruções em ordem de avaliação causadas pelo uso de [illegible].

No Capítulo 10, discutimos métodos através dos quais podemos descobrir que [illegible] poderia apontar somente para alguns segmentos de identificadores. Se [illegible] só pode apontar para [illegible], então [illegible] não precisam ser mortos. É igualmente concebível que poderemos descobrir que [illegible] só pode apontar para um bloco-GDA em que [illegible] só pode ser morto por [illegible]. Entretanto, todos esses tipos de descobertas são complexas e complicadas.

Uma chamada de procedimento para [illegible] termina todos os blocos básicos, uma vez que [illegible] pode ter efeitos colaterais sobre qualquer variável. Isso significa que, [illegible] precisamos considerar que qualquer variável possa ser modificada por uma chamada de [illegible]. Dessa forma, [illegible] precisa ser morto por [illegible]. Entretanto, [illegible] não precisa ser morto por [illegible]. Entretanto, todos esses tipos de descobertas não compensam a complicação.

Uma chamada de procedimento para [illegible] termina todos os blocos básicos, uma vez que [illegible] pode ter efeitos colaterais sobre qualquer variável. Isso significa que, [illegible] precisamos considerar que qualquer variável possa ser modificada por uma chamada de [illegible]. Dessa forma, [illegible] precisa ser morto por [illegible]. Entretanto, todos esses tipos de descobertas não compensam a complicação.

Se pretendermos remontar o GDA em um bloco básico e não desejarmos usar a forma tripla [illegible] do GDA, precisamos, então, indicar no GDA que certos nós apenas imediatamente dependentes precisam ser avaliados em uma determinada ordem. Por exemplo, em [illegible], o comando [illegible] precisa seguir a [illegible] que precisa seguir a [illegible]. Vamos arbitrariamente ligar nós no GDA de forma a que o mesmo resultado ocorra independentemente de nossa ordem de avaliação. Isso é importante porque [illegible] em qualquer computador. As regras que devem ser seguidas são as seguintes:

1. Qualquer atribuição [illegible] precisa seguir a avaliação [illegible] de qualquer [illegible] precisa seguir a avaliação prévia de [illegible].

2. Qualquer [illegible] precisa seguir qualquer avaliação prévia de [illegible].

3. Qualquer uso de qualquer identificador [illegible] precisa seguir uma chamada prévia de procedimento [illegible] através de um apontador, se existir algum.

4. Qualquer chamada de procedimento [illegible] através de um apontador precisa seguir todas as avaliações prévias de qualquer identificador.

Isso é, [illegible] reordenar o código, os usos de um [illegible] não poderão atravessar nós que contêm [illegible] que pode atravessar um chamada de procedimento [illegible] através de um apontador.

9.9 OTIMIZAÇÃO PEEPHOLE

Uma estratégia de geração de código simplesmente examinando frequentemente produz um código ineficiente, contendo instruções redundantes e construções subótimas. A qualidade de tal código pode ser melhorada através da aplicação de transformações [illegible] sobre [illegible] programa-alvo. O termo "otimização" é um tanto enganador porque não existe garantia de o código-fonte resultante ser ótimo segundo qualquer medida matemática. Apesar de tudo, muitas transformações simples podem melhorar significativamente o tempo de execução ou a exigência de espaço do programa-alvo. Dessa forma, é importante saber que tipos de transformações são importantes na prática.

Uma técnica simples, porém eficaz para melhorar [illegible] é a otimização *peephole*, um método para tentar melhorar o desempenho do programa-alvo examinando-se pequenas sequências de instruções-alvo (talvez cada uma de cada vez) e substituindo-se essas instruções por uma sequência mais curta ou mais rápida sempre que possível. Apesar de chamadas otimização *peephole*, essa técnica pode melhorar a qualidade do código-alvo [illegible] pode ser aplicada, diretamente depois da geração do código intermediário [illegible] melhorar aquela representação.

Uma *redundância* é uma pequena [illegible] que o programador ou código-fonte não precisa ser corrigido, apesar de algumas simplesmente [illegible] que é característica da otimização *peephole* que esta melhoria possa criar oportunidades para outras melhorias adicionais. Em geral, sucessivas passagens repetidas sobre o código podem ser [illegible] de uma vez. Nesta seção daremos os seguintes exemplos de transformações de programa que são características do otimizador *peephole*:

- eliminação de instruções redundantes
- eliminação de fluxo de controle
- simplificações algébricas
- uso de dialetos de máquina

Cargas e Armazenamentos Redundantes

Se [illegible] a sequência de instruções

[illegible]

podemos com razão presumir que [illegible] é igual a [illegible] e que [illegible] pode ser [illegible] da [illegible] se [illegible] for [illegible] e [illegible] não for [illegible] depois de [illegible]. Se [illegible] for [illegible] e [illegible] não for [illegible] depois de [illegible], também podemos eliminar [illegible]. Isso pressupõe que [illegible] sempre é executada imediatamente depois de [illegible], e nesse caso não pode [illegible]. Uma maneira de [illegible] é [illegible] e [illegible] têm que estar no mesmo bloco básico para que essa transformação seja segura.

Infelizmente, uma [illegible] como [illegible] não será gerada se o algoritmo sugerido na Seção 8.6 for usado, pois [illegible] só é [illegible] se [illegible] for [illegible] depois de [illegible]. Isso pode ser [illegible] um algoritmo mais [illegible], como aquele [illegible] na Seção 9.1.

Código Inatingível

Outra oportunidade para a otimização *peephole* é a eliminação de instruções inatingíveis. Uma instrução [illegible] também pode ser [illegible] se nenhum desvio [illegible] puder ser [illegible]. Essa operação pode ser repetida para eliminar uma sequência de instruções. Por exemplo, para fins de depuração, um grande programa pode ter [illegible] de si centenas ou milhares de [illegible] que são [illegible] somente se uma variável estiver ligada. Em C, o código [illegible] pode ser parecer com

[illegible]

Esses [illegible] são [illegible] [illegible] [illegible]

Na representação de código intermediário, a instrução condicional pode ser traduzida como

```
    ...
    if ... goto L1
    ...
L1: imprimir informações de depuração
```

Uma otimização óbvia do tipo *propagação de cópias* seria a eliminação dos testes sobre desvios. Por conseguinte, a instrução que segue o rótulo L1 pode ser substituída por

```
L1: imprimir informações de depuração
```

Quando combinada com a estabelecida otimização de programa morto, a propagação de cópias deveria substituir o código por

```
    ...
    if ... goto L1
    ...
L1: imprimir informações de depuração
```

Como o argumento do primeiro comando de desvio condicional avalia para uma constante `true` que pode ser sub-rotina, a instrução `if` pode ser substituída por um desvio incondicional. Assim, todos os comandos que imprimem informações de depuração são marcados como inatingíveis e podem ser eliminados em uma só vez.

Otimizações do Fluxo de Controle

Os geradores de código intermediário do Capítulo 8 frequentemente produzem desvios para desvios, desvios para desvios condicionais ou desvios condicionais para desvios. Esses desvios desnecessários podem ser eliminados tanto no código intermediário quanto no código alvo pelos tipos de otimização *peephole* que se seguem. Basta substituir a sequência de desvios

```
    goto L1
    ...
L1: goto L2
```

pela sequência

```
    goto L2
    ...
L1: goto L2
```

Se não existirem agora desvios para `L1`, pode então ser possível eliminar a instrução rotulada `L1`. Repetindo-se o processo, um procedimento pode ser reduzido a um desvio incondicional. Similarmente, a sequência

```
    if a < b goto L1
    ...
L1: goto L2
```

pode ser substituída por

```
    if a < b goto L2
    ...
L1: goto L2
```

Finalmente, suponhamos que exista somente um desvio para `L2` e que ele seja precedido por um desvio incondicional. A sequência

```
    goto L1
    ...
L2: a = b + c
```

pode ser substituída por

```
    if a < b goto L2
    ...
L2: a = b + c
```

Acrescente-se o número de instruções em 9.11 e 9.12 seja o mesmo, algumas vezes substituímos o desvio incondicional em 9.12, mas nunca em 9.11. Por conseguinte, o código de 9.12 é superior ao de 9.11 em termos do tempo de execução.

Simplificação Algébrica

Não há limite para a quantidade de simplificações algébricas que podem ser implementadas através de otimização peephole. Sem muito esforço, somente umas poucas identidades algébricas ocorrem com uma frequência suficiente o bastante para justificar que sejam implementadas. Por exemplo, eliminamos com

```
x = x + 0
```

ou

```
x = x * 1
```

sequências produzidas por algoritmos de geração de código intermediário e podem ser eliminadas através da otimização *peephole*.

Redução de Capacidade

A redução de capacidade substitui as operações caras por outras equivalentes, mas baratas, em máquinas alvo. Certas instruções de máquina são consideradas especialmente mais baratas que outras e podem ser usadas como casos especiais de operadores mais caros. Por exemplo, é invariavelmente mais barato implementar como `x * x` do que como uma chamada para uma rotina de exponenciação. A multiplicação em ponto fixo por ou divisão por uma potência de dois é mais barata de implementar como um deslocamento. A divisão por uma constante em ponto flutuante pode ser implementada como uma multiplicação por uma constante, operação que pode ser menos custosa.

Uso de Dialetos de Máquina

A máquina alvo pode ter instruções de *hardware* para implementar certas operações específicas eficientemente. Detectar as situações que permitem o uso dessas instruções pode reduzir o tempo de execução significativamente. Por exemplo, algumas máquinas possuem modos de endereçamento que incrementam ou decrementam registradores. Esses modos de incrementar ou subtrair um de um operando antes ou depois de usar seu valor. O uso desses modos melhora grandemente a qualidade de código que empilhar ou desempilhar dados, como na transposição de parâmetros. Esses modos podem também ser usados durante o código para instruções como `x = x + 1`.

9.10 A GERAÇÃO DE CÓDIGO A PARTIR DE GDAs

Nesta seção, mostraremos como gerar código para um bloco básico a partir de uma representação sob a forma de GDA. A vantagem de se fazer isso é que a partir de um GDA podemos mais facilmente ver exatamente quais valores estão disponíveis para uso em cada ponto da sequência final de computação do código. Quando começamos a partir da sequência linear de três endereços ou de quádruplos, a principal dificuldade nessa discussão é o caso de quando o GDA é uma árvore. Para esse caso, podemos gerar um código que podemos provar ser ótimo sob critérios tais como tamanho de programa ou menor número de computações usadas. Esse algoritmo para geração de código foi mediado a partir de uma árvore e também aplicado a código intermediário em uma forma sintática

Rearranjando a Ordem

Vamos rapidamente considerar como a ordem na qual as computações são feitas pode afetar o custo do código objeto resultante. Consideremos o seguinte bloco básico, cuja representação sob a forma de GDA é mostrada na Fig. 9.18. Encontramos que o GDA é uma árvore.

Note-se que a ordem é aquela que seria naturalmente obtida a partir de uma tradução dirigida pela sintaxe para a expressão [illegible] através do Algoritmo 8.3.

Se gerarmos código para os quádruplos de três endereços, usando o Algoritmo da Seção 8.6, obtemos a sequência de código da Fig. 9.19. Assumimos que dois registradores [illegible] estejam disponíveis e que somente [illegible] está saindo.

Por outro lado, vamos supor que rearranjemos a ordem dos quádruplos de forma que o computo de [illegible] ocorra imediatamente antes do cálculo de [illegible]. como

Então, usando o algoritmo de geração de código da Seção 8.6, obtemos a sequência de código da Fig. 9.20. Note-se, novamente [illegible] estão disponíveis. Através da realização da computação nesta ordem, somos capazes de economizar duas instruções: [illegible] que armazena o valor de [illegible] na inicialização de memória [illegible] e [illegible] que recarrega o valor de [illegible] no registrador [illegible].

Um Ordenamento Heurístico para os GDAs

A razão pela qual o segundo ordenamento é tão melhor é o código foi que o computo de [illegible] passou a seguir imediatamente após o computo de [illegible]

Fig. 9.18 GDA para um bloco básico

```
MOV  a, R0
ADD  b, R0
MOV  c, R1
ADD  d, R1
MOV  R0, t1
MOV  e, R0
SUB  R1, R0
MOV  t3, R1
SUB  R0, R1
MOV  R1, t4
```

Fig. 9.19 Sequência de código

seu operando esquerdo na última vez. Que essa disposição é benéfica deveria estar claro. O algoritmo heurístico seguinte para computar [illegible] procura colocar um registrador para o operando esquerdo de [illegible] e imediatamente antes de [illegible] assegura que este será vivo.

Ao selecionar um ordenamento para os nós de um GDA, estamos somente restringindo-nos a assegurar que a ordem preserve o relacionamento dos nós do GDA. Relembramos da Seção 8.6 que esses relacionamentos podem representar um relacionamento operador-operando de restrições impulsionadas decorrentes de preservação de variáveis chamadas de procedimentos, atribuições de ponteiros ou de apontadores. Por ponto o seguinte algoritmo de ordenamento heurístico que tenta, na medida do possível, fazer a avaliação de um nó ser seguida imediatamente à avaliação do argumento mais à esquerda. O algoritmo da Fig. 9.21 produz o ordenamento da ordem inversa.

Exemplo 9.11 O algoritmo da Fig. 9.21 aplicado à árvore da Fig. 9.18 produz a ordem a partir da qual o código da Fig. 9.20 foi produzido. Para um exemplo mais completo, consideremos o GDA da Fig. 9.22.

Inicialmente, [illegible] contém todos os nós interiores. Uma vez que não possuem pais, [illegible] contém [illegible] e [illegible]. Agora, o argumento à esquerda de [illegible] é [illegible]. Assim, [illegible] é selecionado para ser [illegible]. Os nós [illegible] são removidos de [illegible] e [illegible] é colocado em [illegible]. Agora, a linha 14 é selecionada. [illegible] contém [illegible] e [illegible], que possuem um operando esquerdo [illegible]. Consequentemente, selecionamos um nó arbitrário, [illegible] e [illegible] como nosso candidato. Estamos, então, em segunda passagem pelo laço [illegible]. Encontramos [illegible] à esquerda [illegible] e [illegible] [illegible] contém somente [illegible] entre os nós interiores, assim [illegible] é selecionado. A lista resultante [illegible] e [illegible] seguintes, a ordem sugerida de avaliação é [illegible]. Este ordenamento corresponde à sequência de instruções de três endereços

```
MOV  c, R0
ADD  d, R0
MOV  e, R1
SUB  R0, R1
MOV  a, R0
ADD  b, R0
SUB  R1, R0
MOV  R0, t4
```

Fig. 9.20 Sequência de código otimizada

```
(1)   enquanto existirem nós interiores não listados faça início
(2)       selecionar um nó n não listado do qual todos os pais
              já tenham sido listados;
(3)       listar n;
(4)       enquanto o filho mais à esquerda de n tiver todos os
              pais listados e não for ainda filho mais à
              direita de seu pai faça
              { comentário: n deve ser listado antes de seu filho ainda não
              listado }
              início
(5)               listar n;
(6)               n := m
              fim
      fim
```

Fig. 9.21 Algoritmo de Sethi-Ullman.

```
(13)      se n é uma folha então
(12)          se n é o filho mais à esquerda de seu pai então
(13)              rótulo(n) := 1
(14)          senão rótulo(n) := 0
          senão início { n é um nó interior }
(15)          seja m1, ..., mk os filhos de n ordenados por
                  rótulo de tal forma que
                  rótulo(m1) ≥ rótulo(m2) ≥ ... ≥ rótulo(mk);
(16)              rótulo(n) := max(rótulo(mi) + i - 1,
                                   1 ≤ i ≤ k)
          fim
```

Fig. 9.23 Computação de rótulos.

agora gera um código ótimo para a GDA de nossa máquina, seja qual for o número de registradores, se o algoritmo de geração de código da Seção 9.6 for usado. Deve-se ressaltar que, neste exemplo, nossa ordenação heurística nunca teve qualquer escolha para fazer no passo (2) mas, em geral, pode ter muitas escolhas.

Ordenamento Ótimo para Árvores

Segue-se que, para o modelo de máquina da Seção 9.2, podemos fornecer um algoritmo simples para determinar a ordem (lista) na qual todos os enunciados de um bloco básico quando a representação dos mesmos for uma árvore. Ordenar uma árvore significa que, para cada interior, especificamos uma ordem para as subárvores de seus filhos. Uma árvore ordenada é uma árvore na qual os filhos de cada nó interior são ordenados. Um nó é rotulado 1 se for uma folha à esquerda. Um nó é rotulado 0 se for uma folha à direita. Um nó interior é rotulado 1 porque os rótulos de seus filhos são designados e o rótulo mais alto de um filho é 1. A Fig. 9.24 mostra a árvore rotulada resultante. A mesma estabelece que dois registradores são necessários para avaliar a árvore de tal modo que nenhum registrador precise ser armazenado.

Geração de Código a Partir de uma Árvore Rotulada

Apresentamos agora o algoritmo que recebe como entrada uma árvore rotulada T e produz como saída uma sequência de código de máquina que avalia T em r registradores, para qualquer r suficientemente grande. Assumimos que T possui somente operadores binários. A generalização para operadores com um número arbitrário de operandos não é difícil e é deixada como um exercício.

O algoritmo usa a função `gerar_código(n, p, r)` para produzir o código de máquina apropriado para avaliar a subárvore de T com raiz n em r registradores. Depois, também gera a chamada `pilha_r` para alocar registradores. Inicialmente, `pilha_r` contém todos os registradores disponíveis, os quais assumimos que sejam $r-1, r-2, \ldots, 1, 0$ nesta ordem. Uma chamada para `gerar_código` deve ter em conta um subconjunto de registradores, talvez mais pequeno do que o em `pilha_r`. Quando `gerar_código` retorna, deseja os registradores em `pilha_r` na mesma ordem em que os encontrou. O código gerado somente computa o valor da árvore T num registrador no topo de `pilha_r`.

A função `gerar_código(n, p, r)` opera, então, com os dois registradores no topo de `pilha_r`. O uso de `p` nos diz se o filho à esquerda e seu pai se encontram no mesmo registrador.

O procedimento `gerar_código` usa a pilha `pilha_r` para alocar as localizações temporárias de memória. Assumimos que `pilha_r` contém

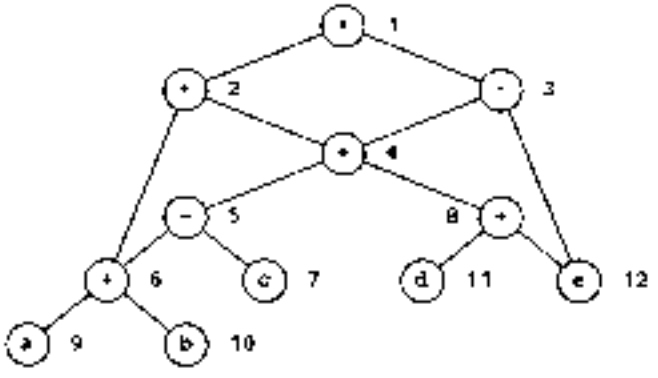


Fig. 9.22 Árvore T.

A rotulação pode ser feita de modo semelhante, de modo que todos os seus filhos tenham sido rotulados. A ordem na qual os nós da árvore gramatical são rotulados será também a ordem na qual o código intermediário, e nesse caso, que será gerado. Os rótulos podem ser computados com uma travessia ascendente pela árvore. A Fig. 9.23 é um algoritmo para computar o rótulo de um nó n. Sua importância especial emerge quando a árvore é binária e seus filhos também rotulados p e q. A fórmula a ser aplicada é

$$rótulo(n) = \begin{cases} \max(p, q) & \text{se } p \neq q \\ p + 1 & \text{se } p = q \end{cases}$$

Exemplo 9.12 Consideremos a árvore da Fig. 9.19. Uma travessia pós-ordem[1] da árvore visita os nós na ordem $a, b, +, c, d, -, *, e, +$. A pós-ordem é especialmente importante, pois a ordem na qual os nós são computados é exatamente a ordem na qual as instruções são geradas. O nó a é rotulado 1 porque é uma folha à esquerda. O nó b é rotulado 0 porque é uma folha à direita. O nó $+$ é rotulado 1 porque os rótulos de seus filhos são designados e o rótulo mais alto de um filho é 1. A Fig. 9.24 mostra a árvore rotulada resultante. A mesma estabelece que dois registradores são necessários para avaliar a árvore de tal modo que nenhum registrador precise ser armazenado.

Geração de Código a Partir de uma Árvore Rotulada

Apresentamos agora o algoritmo que recebe como entrada uma árvore rotulada T e produz como saída uma sequência de código de máquina que avalia T em r registradores, para qualquer r suficientemente grande. Assumimos que T possui somente operadores binários. A generalização para operadores com um número arbitrário de operandos não é difícil e é deixada como um exercício.

O algoritmo usa a função `gerar_código(n, p, r)` para produzir o código de máquina apropriado para avaliar a subárvore de T com raiz n em r registradores. Depois, também gera a chamada `pilha_r` para alocar registradores. Inicialmente, `pilha_r` contém todos os registradores disponíveis, os quais assumimos que sejam $r-1, r-2, \ldots, 1, 0$ nesta ordem. Uma chamada para `gerar_código` deve ter em conta um subconjunto de registradores, talvez mais pequeno do que o em `pilha_r`. Quando `gerar_código` retorna, deseja os registradores em `pilha_r` na mesma ordem em que os encontrou. O código gerado somente computa o valor da árvore T num registrador no topo de `pilha_r`.

A função `gerar_código(n, p, r)` opera, então, com os dois registradores no topo de `pilha_r`. O uso de `p` nos diz se o filho à esquerda e seu pai se encontram no mesmo registrador.

O procedimento `gerar_código` usa a pilha `pilha_r` para alocar as localizações temporárias de memória. Assumimos que `pilha_r` contém

[1] Também se pode pensar nesta travessia pós-ordem como a que é feita por uma função recursiva que visita os filhos de um nó em ordem da esquerda para a direita e depois visita o próprio nó.

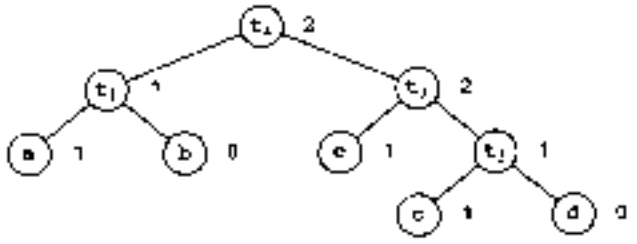
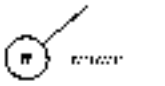


Fig. 9.24 Árvore rotulada

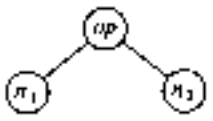
[illegible] inicialmente [illegible]. Na prática, *pilha_r* não precisa ser implementada como uma lista, se simplesmente controlarmos o valor específico de *r* tal que *T* esteja corretamente no topo. O conteúdo da *pilha_r* é sempre um sufixo de [illegible].

O comando X = *desempilhar*(*pilha*) significa "remover o topo da pilha *pilha* e atribuir o valor removido a X". Por outro lado, a ativação *empilhar*(*pilha*, X) para significa "empilhar X na pilha *pilha*". Após *pilha* ser refeita ao valor no topo de *pilha*.

O algoritmo de geração de código chama *gerar_código* à raiz *T*. A rotina *gerar_código* é apresentada na Fig. 9.25. Pode ser explicado através do exame de cada um dos três casos. Para o caso 1 temos uma subárvore da forma



Isto é, o filho é a folha mais à esquerda de seu pai. Por conseguinte, podemos evidentemente cogitar a instrução de carga. Para o caso 2 temos uma subárvore da forma



para a qual queremos código para avaliar n_1 no registrador R = *topo*(*pilha_r*), seguido pela instrução *op* *rótulo* R. No caso 3, temos uma subárvore da forma

onde n_2 pode ser avaliado sem os armazenamentos, mas n_1 é mais difícil de avaliar visto é requer mais registradores do que n_2. Para esse caso, intercambiamos os dois registradores de topo em *pilha_r* e, em seguida, avaliamos n_1 em R = *topo*(*pilha_r*). Removemos R da *pilha_r* e avaliamos n_2 em S = *topo*(*pilha_r*). Note-se que S era o registrador imediatamente abaixo do topo da *pilha_r* no início do caso 3. Geramos, em seguida, a instrução *op* R, S, que produz o valor de n no nó rotulado por *op* no registrador S. Uma outra chamada para *trocar* de *pilha_r* coloca essa situação de volta ao que era antes.

O caso 3 é simétrico ao caso 2, exceto que aqui a árvore à esquerda é mais difícil e é avaliada primeiro. Não há necessidade de se trocar os registradores aqui.

O caso 4 ocorre quando ambas as subárvores requerem *r* ou mais registradores para avaliar sem armazenamentos. Como precisamos de uma localização de memória temporária, avaliamos primeiro a subárvore à direita e armazenamos o resultado em *t*, em seguida, a subárvore à esquerda e, finalmente, a raiz.

Exemplo 9.13 Vamos gerar código para a árvore rotulada da Fig. 9.24 com *pilha_r* = (1, 2, 3, ...) inicialmente. A sequência de chamadas para *gerar_código* e os passos de impressão são mostrados na Fig. 9.26. Mostrado junto aos vértices está o conteúdo de *pilha_r* na [illegible] chamada, com o topo exibido na extremidade direita. A sequência de código aqui é uma permutação daquela da Fig. 9.20. □

Podemos provar que *gerar_código* produz um código ótimo para as expressões de nosso modelo de máquina, assumindo que tenhamos as propriedades algébricas dos operadores que levaram a correção e assumindo que não haja subexpressões comuns. A prova, de [illegible] um exercício, está baseada em se mostrar que qualquer sequência de código ótima que realiza

1. uma operação para cada nó interno

```
procedimento gerar_código(n);
início
  /* caso 0 */
  se n for uma folha à esquerda representando um operando mais à
        esquerda de seu pai então
      imprimir "LOAD " | rótulo(n) | topo(pilha_r);
  senão se n for um nó interno com operador op, filho à esquerda
        n1 e filho à direita n2 então

    /* caso 1 */
    se rótulo(n2) é folha então início
        seja nome o operando representado por n2;
        gerar_código(n1);
        imprimir op | rótulo(n) | "," | nome | "," | topo(pilha_r);
    fim

    /* caso 2 */
    senão se 1 <= rótulo(n2) < rótulo(n1) e rótulo(n2) < r
    então início
        trocar(pilha_r);
        gerar_código(n2);
        R := desempilhar(pilha_r);
        /* n2 foi avaliado no registrador R */
        gerar_código(n1);
        imprimir op | R | "," | topo(pilha_r);
        empilhar(pilha_r, R);
        trocar(pilha_r);
    fim

    /* caso 3 */
    senão se 1 <= rótulo(n1) < rótulo(n2) e rótulo(n1) < r
    então início
        gerar_código(n2);
        R := desempilhar(pilha_r);
        /* n2 foi avaliado no registrador R */
        gerar_código(n1);
        imprimir op | topo(pilha_r) | "," | R;
        empilhar(pilha_r, R);
    fim

    /* caso 4, quando os rótulos >= r, o número total de registradores */
    senão início
        gerar_código(n2);
        T := desempilhar(pilha_r);
        imprimir "STORE " | topo(pilha_r) | "," | T;
        gerar_código(n1);
        empilhar(pilha_r, T);
        imprimir op | T | "," | topo(pilha_r)
    fim
fim
```

Fig. 9.25 Algoritmo gerar_código

```
gerar_codigo(...)                                  /* caso 2 */
    gerar_codigo(...)                              /* caso 3a */
    gerar_codigo(...)                              /* caso 4a */
        imprimir(...)
    gerar_codigo(...)                              /* caso 3b */
        gerar_codigo(...)                          /* caso 4b */
            imprimir(...)
        imprimir(...)
        imprimir(...)
    gerar_codigo(...)                              /* caso 1a */
        gerar_codigo(...)                          /* caso 1b */
            imprimir(...)
        imprimir(...)
    imprimir(...)
```

Fig. 9.26 Rotina `gerar_codigo`.

- uma carga para cada folha que seja um filho mais à esquerda de seu pai v
- uma armazenagem para cada nó, salvo para aqueles cujos filhos estejam todos listados imediatamente antes de v.

Como `gerar_codigo` produz exatamente esses passos, é corrigente.

Operações Multi-Registrador

Podemos modificar o algoritmo de rotulação para tratar operações tais como a multiplicação, divisão ou chamadas de função, que normalmente requerem mais de um registrador para serem realizadas. Simplesmente modificamos o passo (2) da Fig. 9.24, o algoritmo de rotulação, de forma que $rotulo(n)$ seja pelo menos o número de registradores requeridos pela operação. Por exemplo, se é assumido que uma chamada de função requer todos os registradores, então podemos fazer $rotulo(n) = r$. Se a multiplicação requer dois registradores, então a regra binária passa a ser

$$rotulo(n) = \begin{cases} \max(2, l_1, l_2 + 1) & \text{se } l_1 \neq l_2 \\ l_1 + 1 & \text{se } l_1 = l_2 \end{cases}$$

onde l_1 e l_2 são os rótulos dos filhos de n.

Infelizmente, esta modificação não garante que um par de registradores esteja disponível para uma multiplicação ou divisão ou para operações de processador múltiplo. Quase sempre é útil usar algumas máquinas e supor que a multiplicação e a divisão requerem dois registradores. Se essa opção for usada em `gerar_codigo`, então `gerar_codigo` deve conter sempre registradores de números ímpar e consecutivos, r e $r+1$, para algum r. Para conseguir isto, os três primeiros desses registradores são designados incluir um par de registradores. Tirando vantagem do fato de que muitas operações são comutativas, podemos frequentemente evitar o rótulo $rotulo(n) = 2$ de `gerar_codigo` no caso em que n é uma chamada. Igualmente, mesmo que $rotulo(n) = 2$, se um dos três registradores consecutivos for r, temos uma chance muito boa de encontrar um par de registradores em algum local de r ou $r+1$.

Propriedades Algébricas

Se quisermos assumir as leis algébricas padrões sobre vários operadores, temos a oportunidade de substituir uma dada árvore T por uma árvore com menos nós para evitar armazenamentos no caso 4 de `gerar_codigo` e com menos folhas à esquerda para evitar as cargas no caso 3b. Por exemplo, se $+$ é normalmente considerado uma operação comutativa, podemos substituir a árvore da Fig. 9.27(a), rotulada de um número de folhas à esquerda e possivelmente reduzido também alguns rótulos

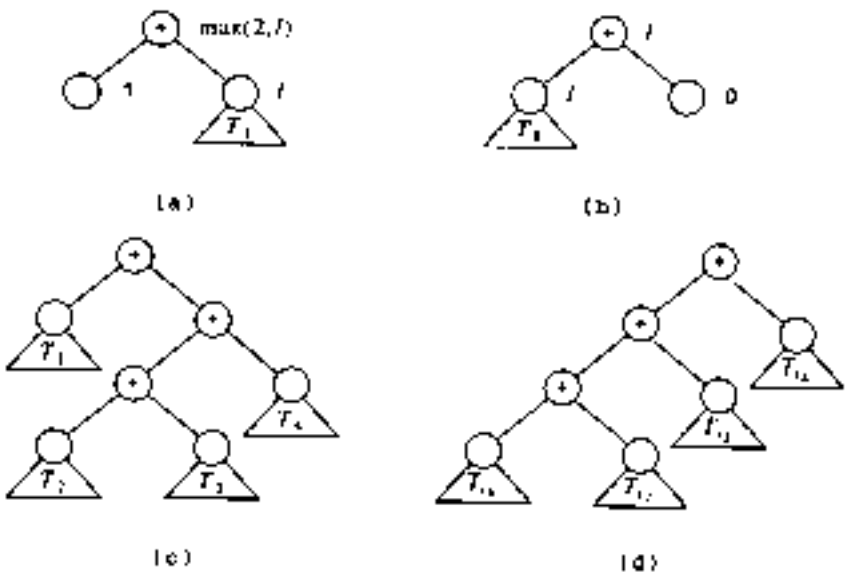


Fig. 9.27 Transformações comutativas e associativas.

Uma vez que a árvore é inteiramente tratada como sendo tanto a sequência que ela computaria, podemos rotular um agrupamento de nós rotulados, como na Fig. 9.27(a), e substituí-lo por uma cadeia à esquerda, como na Fig. 9.27(b). Para minimizar a rotulação da raiz, precisamos somente escolher para que T_1 seja o maior dentre T_1, T_2, T_3 e T_4 tem o maior rótulo e que T_1 não seja uma folha a menos que todos T_1, ..., T_4 o sejam.

Subexpressões Comuns

Quando existem subexpressões comuns num bloco básico, o GDA correspondente já não será mais uma árvore. As subexpressões comuns irão corresponder aos nós com mais de um pai, chamados de nós compartilhados. Já não podemos aplicar mais o algoritmo de rotulação ou o de geração de código diretamente. De fato, as subexpressões comuns tornam a geração de código marcadamente mais difícil do ponto de vista matemático. Bruno e Sethi [1976] mostraram que a geração ótima de código para GDAs numa máquina com um registrador é um problema NP-completo. Aho, Johnson e Ullman [1977] mostraram que mesmo com um número ilimitado de registradores o problema permanece NP-completo. A dificuldade emerge ao se tentar determinar uma ordem ótima onde avaliar um GDA da forma mais barata.

Na prática, podemos obter uma solução razoável se particionarmos o GDA num conjunto de subárvores, procurando, para cada raiz e cada nó compartilhado, a subárvore máxima que tenha o nó como raiz e que não inclua outros nós compartilhados, a não ser como folhas. Por exemplo, o GDA da Fig. 9.22 pode ser particionado nas árvores da Fig. 9.28. Cada nó compartilhado com p pais aparece como uma folha em pelo menos p árvores. Os nós com mais de um pai na mesma árvore podem ser transformados em tantas árvores quantas forem necessárias, de forma que cada uma folha possua múltiplos pais.

Uma vez que particionamos os GDAs em árvores dessa forma, podemos ordenar a avaliação das árvores e usar qualquer um dos algoritmos precedentes para gerar o código para cada árvore. A ordem das árvores precisa ser tal que os valores compartilhados que sejam folhas de uma árvore tenham que estar disponíveis quando a árvore for avaliada. As quantidades compartilhadas podem ser computadas e armazenadas na memória (ou mantidas em registradores, caso haja registradores disponíveis em número suficiente). Conquanto esse processo não gere necessariamente código ótimo, o mesmo será frequentemente satisfatório.

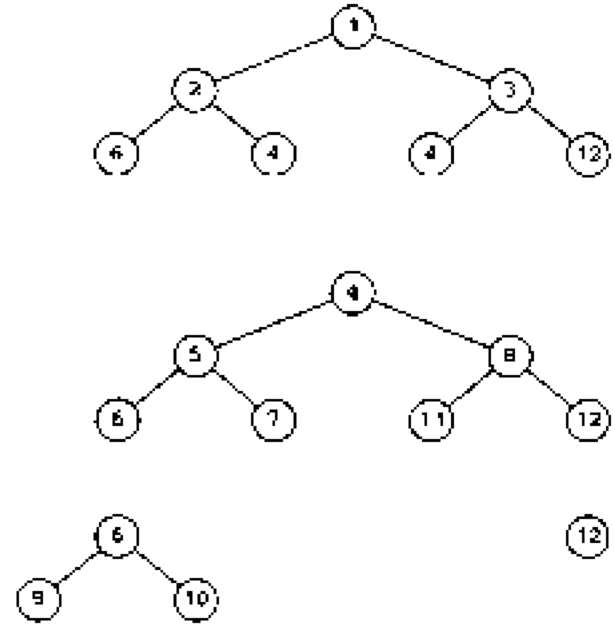


Fig. 9.28 Particionando em árvores.

9.11 ALGORITMO DE PROGRAMAÇÃO DINÂMICA PARA A GERAÇÃO DE CÓDIGO

Na seção anterior, o procedimento *gerar_código* produz um código ótimo a partir de uma árvore de expressão usando uma quantidade de tempo que era uma função linear do tamanho da árvore. Esse procedimento funciona para máquinas nas quais toda a computação seja feita em registradores e cujas instruções consistam em um operador aplicado a dois registradores ou a um registrador e uma localização de memória.

Um algoritmo baseado no princípio da programação dinâmica pode ser usado para estender a classe de máquinas para as quais um código ótimo [illegible] pode ser gerado a partir de árvores de expressões num tempo linear. O algoritmo de programação dinâmica se aplica a uma ampla classe de máquinas de registradores com conjuntos complexos de instruções.

A Classe das Máquinas de Registradores

O algoritmo de programação dinâmica pode ser usado para gerar código para qualquer máquina com r registradores intercambiáveis R0, R1, ..., Rr−1 e instruções da forma $R_i := E$, onde E é uma expressão contendo operadores, registradores e localizações de memória. Se E envolve um ou mais registradores, R_i terá de ser um desses registradores. Esse modelo de máquina inclui o modelo de máquina introduzido na Seção 9.2.

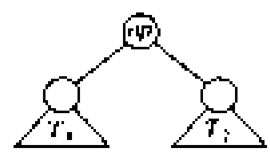
Por exemplo, a instrução [illegible] seria representada como [illegible] + [illegible]. A instrução [illegible] seria representada como [illegible] = [illegible], onde [illegible] está em lugar do operador de indireção.

Assumimos que a máquina possua uma instrução de carga $R_i := M$, uma instrução de armazenamento $M := R_i$ e uma instrução de cópia registrador a registrador $R_i := R_j$. Por uma questão de simplicidade, também assumimos que cada instrução custe uma unidade, apesar do algoritmo de programação dinâmica poder ser facilmente modificado para funcionar mesmo que cada instrução tenha o seu próprio custo.

O Princípio da Programação Dinâmica

O algoritmo de programação dinâmica particiona o problema de se gerar um código ótimo para uma expressão em subproblemas para gerar código ótimo para as subexpressões de dada expressão. Como um exemplo simples, consideremos uma expressão E da forma $E_1 + E_2$. Um programa ótimo para E é formado pela combinação de programas ótimos para E_1 e E_2, numa ou noutra ordem, seguido pelo código para avaliar o operador +. Os subproblemas de se gerar código ótimo para E_1 e E_2 são resolvidos similarmente.

Um programa ótimo produzido pelo algoritmo de programação dinâmica possui uma propriedade importante. Avalia uma expressão $E = E_1$ op E_2 "contiguamente". Podemos apreciar o que isso significa examinando a árvore sintática T para E:



Aqui T_1 e T_2 são árvores para E_1 e E_2, respectivamente.

Avaliação Contígua

Dizemos que um programa P avalia uma árvore contígua T se P primeiro avalia, na memória, aquelas subárvores de T que necessitam ser computadas. Em seguida, avalia o restante de T na ordem T_1, T_2, e, em seguida, a raiz (se T_1 e T_2 estiverem em R, então a raiz em qualquer caso usando os valores previamente computados a partir da memória sempre que necessário). Como um exemplo de avaliação contígua, P poderia primeiro avaliar toda de T_1, deixando o valor num registrador R (e também na memória), em seguida avaliar T_2, e, então, retornar para avaliar o nodo de T_1.

Para a maioria de registradores de frente a uma pilha, provaremos que, dado qualquer programa em linguagem de máquina P para avaliar uma árvore de expressão T, podemos encontrar um programa equivalente P', tal que

1. P' é de custo não-maior do que P.
2. P' não usa mais registradores do que P.
3. P' avalia a árvore numa forma contígua.

O resultado implica que cada árvore de expressão pode ser avaliada otimamente por um programa contíguo.

Para fazermos as comparações entre máquinas com pares ou mais de registradores, tais como as IBM System/370, não possuem sempre avaliações contíguas ótimas. Para as máquinas, podemos usar árvores de expressão para construir um programa otimal em linguagem de máquina, primeiro avaliando numa registrador uma parte da subárvore à esquerda da raiz, em seguida uma parte da subárvore à direita, então outra parte da subárvore à esquerda, a seguir outra parte da subárvore à esquerda e assim por diante. Esse tipo de intercalação é desnecessário para uma avaliação ótima de qualquer árvore de expressão usando a máquina geral de custo dois.

Comprovaremos a avaliação contígua de modo a mostrar que, para qualquer árvore de expressão T, existe sempre um programa ótimo que consiste em programas ótimos para as subárvores da raiz. Essa propriedade nos permite usar o algoritmo de programação dinâmica para gerar um programa ótimo para T.

O Algoritmo de Programação Dinâmica

O algoritmo de programação dinâmica se desenvolve em três fases. Suponhamos que a máquina alvo possua r registradores. Na primeira fase, computamos, bottom-up, para cada nodo da árvore de expressão T um vetor de custos, cujo i-ésimo componente $C(i)$ é o custo ótimo de se computar a subárvore S enraizada por esse nodo, usando i registradores disponíveis para a computação, para $i = 1, 2, \ldots, r$. Os vetores incluem também informações necessárias para se obter S no número ótimo de registradores. Inclui também o custo de computar o operador da raiz de S. O r-ésimo componente do vetor de custos é o custo ótimo para se computar a subárvore S na memória. A propriedade da avaliação contígua assegura que um programa ótimo para S pode ser gerado, considerando-se as combinações de programas ótimos somente para as subárvores da raiz de S. Essa restrição reduz o número de casos que precisam ser considerados.

Para computar $C(i)$ em um nodo n, consideramos cada instrução de máquina I em cuja expressão E corresponda à subárvore cuja raiz é o nodo n. Pelo exame dos vetores de custo para os descendentes de n, podemos determinar os custos de se avaliar os operandos de E. Para aqueles operandos de I que sejam registradores, consideramos todas as possíveis ordens nas quais as subárvores correspondentes de T possam ser avaliadas em registradores. Em cada ordenamento, a primeira subárvore corresponderá a um operando registrador pode ser avaliada usando i registradores disponíveis, a segunda usando $i-1$ registradores e assim por diante. Para cada, mais uma vez, adicionamos o custo da instrução I que foi usada para corresponder ao nodo. O valor de $C(i)$ será então mínimo sobre todas as possíveis ordens.

Os custos de custos para a localidade da árvore T pode ser computados de baixo para cima em tempo aproximadamente proporcional ao número de nós de T. Lembremos anteriormente a escolha da instrução usada para atingir o melhor custo para $C(i)$ para cada valor de i. O menor custo no vetor de custos da raiz de T fornece o custo mínimo para se avaliar T.

Na segunda fase do algoritmo, fazemos uma travessia da árvore usando os vetores de custo para determinar que subárvores de T precisam ser computadas na memória. Na terceira fase, fazemos uma travessia de cada árvore usando os vetores de custos e instruções associadas para gerar o código-alvo final. O nível para as subárvores computadas em localizações de memória é gerado primeiro. Essas duas fases podem também ser implementadas de forma a que poderem ser feitas juntamente proporcional ao tamanho da árvore da expressão.

Exemplo 9.14. Consideremos uma máquina que tenha dois registradores e as seguintes instruções, cada uma de custo unitário.

LD r, m	$r = m$
ST m, r	$m = r$
OP r, m	$r = r$ op m
OP r, r	$r = r$ op r
OP r_1, r_2	$r_1 = r_1$ op r_2

Nessas instruções, r e r_1 e r_2 denotam localizações de registradores, e m denota uma localização de memória.

Vamos aplicar o algoritmo de programação dinâmica para gerar um código ótimo para a árvore sintática da Fig. 9.29. Na primeira fase, computamos os vetores de custo mostrados a seguir. Para folha a, o vetor de custo é $(0, 1, 1)$. O custo de computar a na memória é 0, uma vez que já está lá. O custo de computar a em um registrador é 1, uma vez que podemos carregar a em um registrador através da instrução LD. O custo de se computar a em um registrador, tendo dois registradores disponíveis, é o mesmo do que quando se tem um único registrador. O vetor de custos da folha b é, por conseguinte, $(0, 1, 1)$.

Consideremos o vetor de custos da raiz $+$. Primeiro, determinamos o custo mínimo de se computar a raiz com um e dois registradores disponíveis. A instrução de máquina ADD r_1, r_2 se ajusta à raiz, pois que a raiz é rotulada com o operador $+$. Usando esta instrução, o custo mínimo de se avaliar a raiz com um registrador disponível é o custo

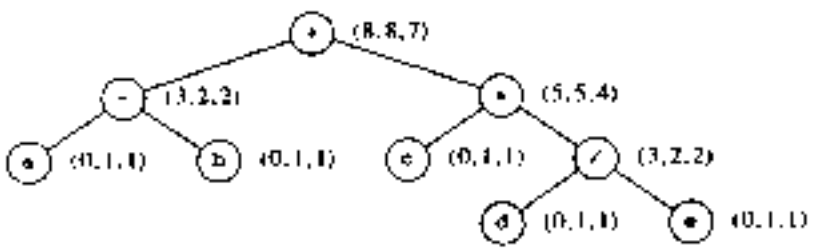


Fig. 9.29 Árvore sintática para [illegible] com vetor de custos avaliados

[illegible] custo de computar uma subárvore direita, com menos registradores [illegible] computar a subárvore à esquerda num registrador [illegible] para armazenar. Nas duas outras [illegible]. Os vetores de custos dos filhos à direita e à esquerda da raiz constante que [illegible] se computar a raiz com um registrador disponível é $5 + 2 + 1 = 8$.

Consideremos agora [illegible] a raiz com dois registradores disponíveis. As alternativas surgem dependendo de que instrução é usada para computar a raiz e em que ordem as subárvores da esquerda e da direita, de raiz, são avaliadas.

1. Computar a subárvore à esquerda no registrador [illegible] com dois registradores disponíveis e computar a subárvore à direita no registrador [illegible], com um registrador disponível, e usar a instrução [illegible] para computar a raiz. O custo desta sequência é $2 + 5 + 1 = 8$.

2. Computar a subárvore à direita com dois registradores disponíveis em [illegible], computar a subárvore à esquerda com um registrador disponível em [illegible] e usar a instrução [illegible]. Essa sequência custa também $4 + 2 + 1 = 7$.

3. [illegible] computar a subárvore à direita na localização de memória [illegible], computar a subárvore à esquerda com dois registradores disponíveis e usar a instrução [illegible]. Essa sequência possui custo $5 + 2 + 1 = 8$.

A segunda escolha fornece o custo mínimo.

O custo mínimo [illegible] computar a raiz na memória e determinado pela adição de um [illegible] mínimo de se computar a raiz com todos os registradores disponíveis, mais [illegible] computar a raiz num registrador e, em seguida, armazenar o resultado. O custo de custos a raiz é, por conseguinte, $\min(8, 8, 7) = 7$.

A partir dos vetores de custos, podemos construir facilmente a sequência de código fazendo-se uma travessia da árvore. A [illegible] Fig. 9.29, pressupondo que dois registradores estejam disponíveis, uma sequência ótima é

[illegible]

Essa técnica, originalmente descrita por Aho e Johnson [1976], tem sido usada em vários compiladores, incluindo a segunda versão do SCC, [illegible] para um computador Cyber, e o PCC2. A principal falha da técnica é a suposição [illegible] que causa [illegible] da técnica de programação orientada a uma ampla classe de máquinas.

9.12 GERADORES DE GERADORES DE CÓDIGO

A geração de código envolve a escolha de uma ordem de avaliação para as operações, a alocação dos registradores para a guarda de valores e a seleção das instruções apropriadas da linguagem-alvo para implementar os operadores da representação intermediária. Mesmo que essas decisões sejam tomadas [illegible] de avaliação fixos fornecidos e os registradores alocados por um [illegible] separado, o problema de decidir quais instruções usar pode ser uma grande tarefa combinatorial, especialmente numa máquina [illegible] ortogonais. Nesta seção, apresentamos técnicas de reescrita de árvores que podem ser usadas para construir automaticamente a fase de seleção de instruções de um gerador de código, a partir de uma especificação simbólica [illegible] da máquina-alvo.

Geração de Código através da Reescrita de Árvore

A partir desta seção, [illegible] pela representação de geração de código será uma sequência de árvores [illegible] nível semântico da máquina-alvo. As árvores são [illegible] que podem ser [illegible] após inserir os endereços em tempo de execução na representação intermediária, como descrito na Seção 9.3.

Exemplo 9.15. A Fig. 9.30 contém uma árvore para [illegible] atribuição [illegible] variáveis [illegible] são nomes locais com endereços em tempo de execução. Os [illegible] nomes desses [illegible] a partir de [illegible] registrador que aponta [illegible] para o início do registro de ativação corrente. Usamos [illegible] armazenar um [illegible] em tempo de execução. A atribuição [illegible] armazenar o valor do lado direito [illegible] o endereço do lado esquerdo. [illegible] constante [illegible] com o símbolo [illegible] adição do valor da constante [illegible] conteúdo do registrador [illegible] valor de [illegible] obtida pela adição do valor da constante [illegible] endereço de registrador [illegible]. A [illegible] é uma [illegible] global com localização de memória [illegible]. Por uma questão de simplicidade, [illegible] que todas as variáveis [illegible] do tipo caractere.

Nas árvores, [illegible] operador [illegible] dois argumentos: o da esquerda [illegible] uma localização de memória. Como filho à esquerda de um operador de atribuição, o [illegible] fornece a localização [illegible] calcular o lado direito do operador de atribuição [illegible] deve ser armazenado. Se o argumento à esquerda do operador [illegible] for uma localização de memória ou um registrador, o conteúdo daquela localização de memória ou registrador é [illegible] nodos [illegible] valor. As folhas da árvore são atributos de [illegible] valores [illegible] subscrito indica o valor do atributo. [illegible]

O código-alvo é gerado durante [illegible] processo [illegible]. A [illegible] de entrada é [illegible] uma árvore [illegible] através da aplicação de uma [illegible] sequência de regras de reescrita de árvores. Cada regra de reescrita de árvore é [illegible] da forma

$$\text{substituição} \longrightarrow \text{subárvore} \quad \{\text{ação}\}$$

onde

1. *substituição* é um único nó,
2. *subárvore* é uma árvore, e
3. *ação* é um fragmento de código, [illegible] a ser executado [illegible] regra pelo [illegible].

Um [illegible] de regras de reescrita de árvores é chamado de *esquema de tradução de árvores*.

Cada gabarito de árvore representa uma [illegible] pela sequência de instruções de [illegible] e [illegible] pela [illegible] associada. Especificamente, um gabarito corresponde a uma única instrução de máquina. As folhas dos gabaritos são [illegible] com subscritos [illegible]

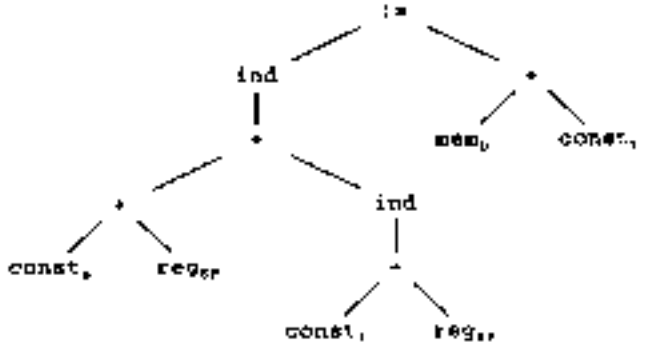


Fig. 9.30 Árvore de código intermediário para [illegible] = [illegible]

árvore de entrada. Frequentemente, certas restrições se aplicam aos valores dos subscritos nos gabaritos; essas restrições são especificadas como predicados semânticos que precisam ser satisfeitos antes de gabarito ser considerado como correspondente. Por exemplo, um predicado poderia especificar que o valor de uma constante fosse zero ou determinado número.

Um esquema de tradução de árvores é uma forma conveniente de se representar a fase de seleção de instruções de um gerador de código. Como exemplo de uma regra de reescrita de árvores, consideremos a regra para a instrução de adição registrador a registrador,

$$reg_i \leftarrow +(reg_i, reg_j) \quad \{ \texttt{ADD } R_j, R_i \}$$

Esta regra é lida como se segue. Se a árvore de entrada contém uma subárvore que se ajusta a esse gabarito de árvore, isto é, uma subárvore cuja raiz seja rotulada pelo operador + e cujos filhos à esquerda e à direita sejam quaisquer nós registradores reg_i e reg_j, respectivamente, aquela subárvore pode ser substituída por um único nó rotulado reg_i. A instrução $\texttt{ADD } R_j, R_i$ é emitida como saída. Mais de um gabarito pode se ajustar a uma subárvore a um dado instante; iremos descrever em breve algumas heurísticas para decidir que regra aplicar no caso de um conflito. Assumimos que a alocação de registradores seja feita antes da seleção de código.

Exemplo 9.16 A Fig. 9.31 contém regras de reescrita de árvores para uma pequena instrução em nossa máquina alvo. Essas regras serão usadas em um exemplo que percorreremos ao longo desta seção. As duas

(1)	reg_i	$\leftarrow const_c$	{ MOV #c, Ri }
(2)	reg_i	$\leftarrow mem_a$	{ MOV a, Ri }
(3)	mem	$\leftarrow$ = (mem_a, reg_i)	{ MOV Ri, a }
(4)	mem	$\leftarrow$ = (ind(reg_i), reg_j)	{ MOV Rj, *Ri }
(5)	reg_i	$\leftarrow$ ind(+($const_c$, reg_j))	{ MOV c(Rj), Ri }
(6)	reg_i	$\leftarrow$ +(reg_i, ind(*($const_c$, reg_j)))	{ ADD c(Rj), Ri }
(7)	reg_i	$\leftarrow$ +(reg_i, reg_j)	{ ADD Rj, Ri }
(8)	reg_i	$\leftarrow$ +(reg_i, $const_1$)	{ INC Ri }

Fig. 9.31 Regras de reescrita de árvores para instruções de uma máquina alvo.

[illegible] regras correspondem a instruções de carga, as duas seguintes a operações de armazenamento e o restante a adições e cargas indexadas. Note-se que a regra (6) requer que o valor do registrador seja mantido. Essa condição será especificada por um predicado semântico.

Um esquema de tradução de árvores funciona da seguinte forma. Dada uma árvore de entrada, os gabaritos das regras de reescrita de árvores são aplicados às sub-árvores. Se a ação contém uma sequência de instruções de máquina, as instruções são emitidas. Este processo é repetido até que a árvore seja reduzida a um único nó ou até que nenhum dos gabaritos possa ser aplicado. A sequência de instruções de máquina gerada é tal que a árvore de entrada era reduzida a um único nó se construído a partir de um esquema de tradução de árvores da árvore de entrada.

O processo de especificar um gerador de código pode ser semelhante àquele de se usar um esquema de tradução dirigida pela sintaxe para especificar um tradutor. Escrevemos um esquema de tradução de árvores para descrever o conjunto de instruções de uma máquina alvo. Na prática, nos esforçamos de encontrar um esquema que fornece código gerado uma sequência de instruções de custo mínimo para cada árvore de entrada. Várias ferramentas estão disponíveis para auxiliar a construção do gerador de código automaticamente a partir de um esquema de tradução de árvores.

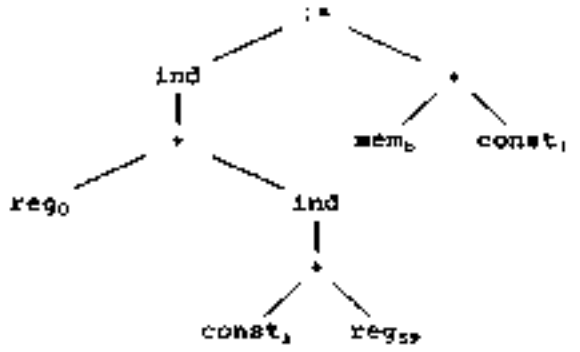
Exemplo 9.17 Vamos usar o esquema de tradução de árvores da Fig. 9.31 para gerar código para a árvore de entrada na Fig. 9.30. Suponhamos que a primeira regra

[illegible]

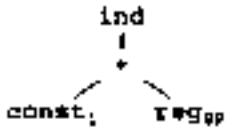
seja aplicada para se carregar a constante 1 no registrador 1. O custo para a folha mais à esquerda também é [illegible] e a instrução [illegible] é gerada. A última regra

(7) reg_0 — (+ reg_0 reg_{SP}) { ADD SP,R0 }

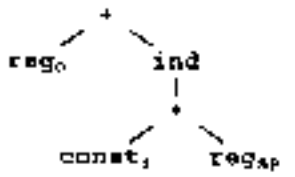
agora se ajusta à sub-árvore para o qual o custo total é [illegible]. Usando esta regra, reescrevemos a sub-árvore como um único nó rotulado [illegible] e geramos a instrução [illegible]. Agora a árvore está como



A esse ponto podemos aplicar a regra (5) para reduzir a sub-árvore

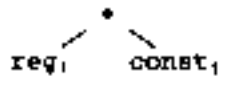


a um único nó rotulado [illegible]. Se ignorarmos por um momento a regra (6) para reduzir a sub-árvore maior num único nó rotulado [illegible] e gerar a instrução [illegible],

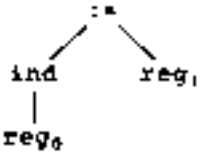


Vemos que seja mais eficiente usar uma única instrução para computar a sub-árvore maior, através da regra, escolhemos a única solução para obter

No sub-árvore à direita, a regra (2) se aplica à folha [illegible]. Esta regra gera uma instrução para carregar [illegible] no registrador 1, digamos. Agora usando a regra (3), podemos empacotar a sub-árvore e gerar a instrução de incremento [illegible]. Nesse ponto, a árvore de entrada se reduz a



Esta árvore pode ser reduzida, pela regra (4), que a reduz em um nó e gera a instrução [illegible].



No processo de redução da árvore, as três instruções geradas são a seguinte sequência de código:

[illegible]

Vários aspectos desse processo de redução de árvores precisam de explicações subsequentes. Não especificamos como reconhecer a ocorrência do padrão de árvores a ser feito. Também especificamos uma ordem na qual os gabaritos são reconhecidos ou o que fazer se mais de um gabarito se ajustar a mesma sub-árvore para um dado nó. Suponhamos, por enquanto, que se escolhamos qualquer gabarito que se ajuste a uma sub-árvore, o processo de geração de código fica bloqueado. Também estamos cientes de que é possível que um único nó seja reescrito indefinidamente, gerando uma sequência infinita de instruções de duplicação de registradores ou uma sequência infinita de cargas e armazenamentos.

Uma forma de se realizar o casamento dos padrões de árvores eficientemente é estender o algoritmo de reconhecimento de cadeias para múltiplas palavras-chave (ver Seção 3.4.2), numa gabarito gerador de reconhecimento de padrões de árvores. Cada gabarito pode ser representado por um conjunto de cadeias de caracteres representando, respectivamente, pelo conjunto de percursos da raiz até as folhas. A partir dessas coleções de cadeias, podemos construir um reconhecedor de padrões de árvores, como no Exercício 9.32.

(1)	$reg_i \rightarrow const_c$	{ MOV #c, Ri }
(2)	$reg_i \rightarrow mem_a$	{ MOV a, Ri }
(3)	$mem \rightarrow := mem_a\ reg_i$	{ MOV Ri, a }
(4)	$mem \rightarrow := ind\ reg_i\ reg_j$	{ MOV Rj, *Ri }
(5)	$reg_i \rightarrow ind + const_c\ reg_j$	{ MOV c(Rj), Ri }
(6)	$reg_i \rightarrow + reg_i\ ind + const_c\ reg_j$	{ ADD c(Rj), Ri }
(7)	$reg_i \rightarrow + reg_i\ reg_j$	{ ADD Rj, Ri }
(8)	$reg_i \rightarrow + reg_i\ const_1$	{ INC Ri }

Fig. 9.32 Esquema de tradução dirigida pela sintaxe do gerador de código.

Os problemas de ordenamento e reconhecimento múltiplo podem ser resolvidos quando do uso do reconhecimento de padrões de árvores em conjunto com o algoritmo de programação dinâmica descrito na seção anterior. Uma esquema de tradução de árvores pode ser expandido com informações de custo, associando-se a cada regra de reescrita de árvores o custo da sequência de instruções de máquina gerada se aquela regra for aplicada.

No processo de reescrita de árvores pode ser implementado rotinas que reconhecem os padrões de árvores durante uma passagem em profundidade da árvore de entrada e realizam as reduções à medida que os nós são visitados pela última vez. Se estamos utilizando o algoritmo de programação dinâmica simultaneamente, podemos selecionar uma sequência ótima de reconhecimentos usando as informações de custo associadas a cada regra. Podemos necessitar postergar a decisão a respeito de um reconhecimento até que os custos de todas as alternativas seja conhecido. Usando esta abordagem, um pequeno e eficiente gerador de código pode ser construído capaz de gerar bom código sequencial de árvores sintáticas. Sobretudo, o algoritmo de programação dinâmica oferece a possibilidade de gerar código de tempo ótimo para árvores de expressão. Infelizmente, os problemas de reconhecimento múltiplo e ordenamento continuam a existir.

Reconhecimento de Padrões Através da Análise Sintática

Uma outra abordagem é usar um analisador sintático LR para realizar o reconhecimento de padrões. A árvore de entrada pode ser transformada em uma cadeia de caracteres através do uso de sua representação prefixa. Por exemplo, a representação prefixa para a árvore da Fig. 9.30(a) é

= , [ ] ... + ... [ ] ...

Do esquema de tradução de árvores pode ser convertido num esquema de tradução dirigida pela sintaxe substituindo as regras de reescrita de árvores por produções de uma gramática livre de contexto na qual os lados direitos sejam representações de prefixos dos padrões de instruções.

Exemplo 9.18 O esquema de tradução dirigida pela sintaxe da Fig. 9.32 está baseado no esquema de tradução de árvores da Fig. 9.31. —

A partir do esquema de tradução construímos um analisador sintático LR usando uma das técnicas de construção do Capítulo 4. O código-alvo é gerado através da execução da ação de máquina correspondente a cada redução.

Uma gramática de geração de código é usualmente ambígua e algumas escolhas devem ser tomadas a respeito de como resolver os conflitos das ações sintáticas se são resolvidos quando o analisador sintático é construído. Na inserção de informações de custo, uma regra geral é de favorecer as reduções maiores e as que englobam mais nós. Isto significa que um conflito reduz-reduz a redução mais longa é preferida. Um conflito empurra-reduz, o mesmo método é empregado e escolhido. Esse método de "escolha gulosa" faz com que um maior número de operações seja realizado por uma única instrução de máquina.

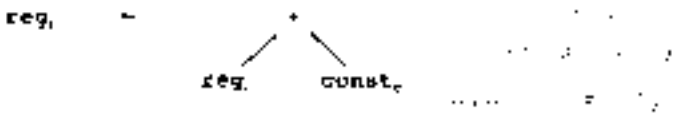
Existem vários aspectos associados à técnica de análise sintática para geração de código. Primeiro, o método de análise sintática é eficiente e bem conhecido e, por conseguinte, geradores de código confiáveis e eficientes podem ser criados usando-se os algoritmos descritos no Capítulo 4. Segundo, e relativamente falando, o esforço para gerar um código para uma nova máquina pode ser consideravelmente reduzido se uma gramática que descreva as instruções da nova máquina. Terceiro, o código gerado pode ser tornado eficiente adequando-se posições apropriadas especiais de forma a explorar vantagens dos detalhes da máquina.

No entanto, existem algumas dificuldades potenciais. Uma ordem de avaliação da esquerda para a direita é ditada pelo método de análise sintática. Infelizmente, para algumas máquinas com um amplo número de métodos de endereçamento, a gramática de descrição da máquina será ambígua e sintaticamente poderá ser muito excessivamente grande. Como consequência, técnicas especializadas são necessárias para se evitar explosões gramaticais de reconhecedores sintáticos. Precisamos também ser cuidadosos para que o analisador sintático resultante não fique bloqueado (não possua próximo movimento) porque poderá ocorrer uma expressão de árvore de entrada que não é gerada pela gramática mas contém alguns padrões de operadores e operandos que o analisador sintático tenha tentado reduzir. Finalmente, algumas dificuldades sintáticas podem ocorrer quando a ordem de avaliação é ditada pela sintaxe. Precisamos também assegurar que o analisador sintático não contenha uma lógica infinita de reduções de produções com símbolos únicos no lado direito. O problema do laço infinito pode ser resolvido usando-se uma técnica de particionamento de estados no tempo em que as tabelas sintáticas são geradas (ver Glanville [1977]).

Rotinas para a Verificação Semântica

As folhas na árvore de entrada são atributos de tipo conhecido, pode um subconjunto associado com cada operador. Não esquema de tradução para geração de código os mesmos atributos aparecem mas, frequentemente, com restrições sobre os valores que os atributos possam ter. Por exemplo, uma instrução de máquina pode requerer que um atributo constante tenha um valor que os valores de deslocamento válidos sejam

Essas restrições sobre os valores de atributos podem ser especificadas como predicados que são avaliados antes de uma redução ser feita. De fato, uma gramática de atributos semânticos pode conter uma flexibilidade e facilidade de descrição ideais para especificação prática para um gerador de código. Os gabaritos gerados podem ser usados para representar classes de instruções e atributos semânticos podem ser usados para selecionar instruções para casos específicos. Por exemplo, duas formas de instrução de adição podem ser representadas como um gabarito:



Os atributos de regras semânticas podem ser resolvidos normalmente, para os predicados que permitem diferentes estratégias de avaliação serem usadas em diferentes contextos. Uma descrição mental disso é possível porque certos aspectos da gramática de atributos, como as árvores usadas de endereçamento, podem ser inseridos nos atributos. A complicação desse método é que pode ser muito difícil estabelecer a correção da gramática como uma descrição fiel da linguagem alvo, apesar do problema ser computacionalmente fácil para os geradores de código.

EXERCÍCIOS

9.1 Gere código para os seguintes enunciados C para a máquina alvo da Seção 9.2, assumindo que todas as variáveis são estáticas. Assuma que três registradores estejam disponíveis.

[illegible]

9.2 Repita o Exercício 9.1 assumindo que todas as variáveis sejam automáticas (reservadas na pilha).

9.3 Gere P-Code para os seguintes enunciados em C para a máquina alvo da Seção 9.2, assumindo que todas as variáveis sejam estáticas. Assuma que três registradores estejam disponíveis.

[illegible]

9.4 Repita o Exercício 9.1 usando:
a) o algoritmo da Seção 9.6
b) o procedimento gerar_código
c) o algoritmo de alocação dinâmica da Seção 9.11

9.5 Gere código para os seguintes enunciados C

[illegible]

9.6 Gere código para o seguinte programa C

[illegible]

9.7 Suponha que para o laço da Fig. 9.13 os pontos para escapar três registradores para alocar [illegible]. Gere o código para os blocos daquele laço. Compare o custo de seu código com o daquele da Fig. 9.14.

9.8 Construa o grafo de interferência de registradores para o programa da Fig. 9.15.

9.9 Suponha que, para uma questão de simplicidade, mantivéssemos automaticamente todos os registradores na pilha em uma chamada de subrotina. Para não ter de salvar cada chamada de procedimento e os registradores após retornos. Como isso influenciaria a fórmula (9.1) usada para calcular a utilidade de se alocar um registrador a uma dada variável dentro de um laço?

9.10 Modifique a função obter_registrador da Seção 9.6 para que forme pares de registradores sempre necessários.

9.11 Dê um exemplo de um GDA para o qual o processo heurístico de ordenação de nós de um GDA, fornecido na Fig. 9.21, não devolve o melhor ordenamento possível.

→9.12 Gere um código ótimo para os seguintes enunciados de atribuição:

[illegible]

9.13 Gere o código para o seguinte programa Pascal:

[illegible]

9.14 Construa o GDA para o seguinte bloco básico:

[illegible]

9.15 Quais são os ordens legais de avaliação para os nomes para os valores nos nós para o GDA do Exercício 9.14
a) assumindo que [illegible] estejam vivos ao final do bloco básico
b) assumindo que somente [illegible] esteja vivo ao final?

9.16 No exercício 9.15(b), se estivermos gerando código para uma máquina com um registrador somente, que ordem de avaliação será a melhor?

9.17 Podemos modificar o algoritmo de construção de GDAs, de forma a lidarmos com casos de [illegible] as atribuições através de apontadores. Quando queremos determinar se um nó pode ser reutilizado em uma atribuição, assumimos que esse nó só será reutilizado para aquele propósito. Esse nó só representará por um momento o valor antigo do [illegible], e o nó do índice para o elemento do vetor [illegible] Quando uma atribuição ocorre através de um apontador, assumimos que criamos um novo valor para cada variável que aquele apontador poderia ter apontado. Os filhos do nó para cada novo valor são o valor do apontador e o valor antigo da variável que poderia ter sido atribuída. Usando essas suposições, construa um GDA para o seguinte fluxo básico:

[illegible]

Assuma que [illegible] possa apontar para qualquer local, [illegible] apenas somente para [illegible]. Não se preocupe de mesmo a área e [illegible] das restrições.

9.18 Se um apontador [illegible] tal como [illegible] e [illegible] atribuído e então usado sem a possibilidade de seu valor ter sido atribuído nesse intervalo, poderíamos reconhecer a situação e fixar a origem dela para simplificar o GDA. Por exemplo, no código do Exercício 9.17, uma vez que [illegible] não é atribuído entre o segundo e quarto enunciados, o enunciado [illegible] pode ser substituído por [illegible], porque estamos seguros de que, qualquer que seja a localização que [illegible] aponte, o mesmo passou

mesmo valor que ... [illegible] ... apenas. Revise o algoritmo de construção de GD's de forma a que haja vantagem de tais ocorrências. Aplique seu algoritmo ao código-objeto do Exercício 9.17.

**9.19 Crie um algoritmo para gerar código-objeto para uma sequência de empilhamentos de três endereços da forma ... [illegible] ... máquina de registradores do Exemplo 9.14. Os empilhamentos têm que ser executados na ordem dada. Quais são a complexidade de tempo de seu algoritmo?

NOTAS BIBLIOGRÁFICAS

Há um interesse em vários gêneros de pesquisa em geração de código-objeto. Deve-se consultar Wulf [1979a,b], Aho e Sethi [1977], Graham [1980] e [1984], Ganapathi, Fischer e Hennessy [1982], Lunell [1983] e Henry [1984]. A geração de código para Bliss é descrita por Wulf et al. [1975], para Pascal por Ammann [1977] e para PL/8 por Auslander e Hopkins [1982].

As estatísticas de uso de programas servem para o projeto de compiladores. Knuth [1971a] fez um estudo empírico de programas Fortran, Satterthwaite [1976] apresenta algumas estatísticas sobre o uso de PL/I e Shustek et al. [1978] e Carter [1982] analisam programas Pascal. O desempenho de vários compiladores sobre diversos conjuntos de instruções é discutido por Lunde [1977], Shustek [1978] e Dalton e McClellan [1982].

Muitos dos processos heurísticos para a geração de código-objeto deste capítulo foram usados em vários compiladores. Leverett et al. [1974] discute as vantagens de usar cores como uma aproximação para a geração de códigos de qualidade para blocos básicos. A estratégia usada em *alocar_reg* para escolher um registrador livre, expulsando se for preciso um registrador cujo valor será usado novamente com menos prioridade, foi mostrada segundo uma variação de busca de páginas por Belady [1966]. Sobre estratégias de alocar um número fixo de registradores para guardar variáveis pela duração de um laço foi mencionado, por Morrill [1970], e estudado na implementação de Fortran H por Lowry e Medlock [1969].

Horwitz et al. [1966] foram os primeiros a usar a noção de registradores índice em Fortran. O relacionamento de gráficos com uma técnica de alocação de registradores foi proposto por Cocke, Kennedy [1971] e Schwartz [1973]. O tratamento de vértices de grau 1 da Seção 9.7 segue Chaitin et al. [1981] e Chaitin [1982]. Chow e Hennessy [1984] descrevem um algoritmo de colorimento de gráficos baseado em prioridades para a alocação de registradores. Outras estratégias para a alocação de registradores são discutidas por Kennedy [1972], Johnson [1974], Harrison [1977], Beatty [1974] e Leverett [1982].

O algoritmo de substituição para árvores da Seção 9.10 é uma extensão do algoritmo para demonstrar que a contribuição de um registrador mais antigo para a instrução mais recente é a contribuição de todos os seus usos, que recebe um novo nome daí em diante. O algoritmo de rotulação apareceu originalmente em Ershov [1958]. Os algoritmos de geração de código que usam este método foram propostos por Anderson [1964], Nievergelt [1965], Nakata [1967], Redziejowski [1969] e Beatty [1972]. Sethi e Ullman [1970] usaram o método de rotulação num algoritmo que foram capazes de provar que gerava um código-objeto ótimo para árvores de expressões numa máquina de registradores. O procedimento geral de geração da Seção 9.10 é uma modificação de algoritmo de Sethi e Ullman, descrito em Stockhausen [1975], Bumgarner, Bussenguth [1975] e Carthwright e Sethi [1983]. Formas mais gerais de geração de código-objeto para árvores de expressões se aplicam a máquinas que possuem registradores que precisam ser tratados como uma pilha.

Aho e Johnson [1976] descrevem o algoritmo de programação dinâmica descrito na Seção 9.11. Este algoritmo foi usado como base para a geração de código no compilador do sistema PCC e foi também usado por Ripken [1977] num compilador para a máquina IBM 370. Knuth [1977] generalizou o algoritmo de programação dinâmica para árvores com custos associados de registradores (ou cores) e IBM [1980] e DEC [1978]. Ao desenvolver a generalização de Knuth, sorgiu a geração de código como um problema de ótimo combinatório para gramáticas livres de contexto.

Floyd [1961] formulou um algoritmo para criar subexpressões comuns em expressões aritméticas. Os desentrelaçamentos de GD's em árvores e o uso de um pré-processador como forma de gerar código separadamente sobre as árvores é proveniente de Wulfs [1975a], Sethi [1975] e Aho e Sethi [1976] investigam que o problema da geração de código ótimo para GD's é NP-completo. Aho, Johnson e Ullman [1977a] mostram que o problema permanece NP-completo com máquinas de operações binárias e de número infinito de registradores. Aho, Ganapathi e Ullman [1977a] e Gonzalez e Johnson [1979] discutem a implementação de geração de código para um problema ser NP-completo.

As várias formulações sobre os blocos básicos foram estudadas por Aho e Ullman [1972a] e por Downey e Sethi [1978]. A otimização peephole é discutida por McKeeman [1965], Fraser [1976], Davidson e Fraser [1980] e Stoutamire [1981] e Gregerich [1983]. Tanenbaum, van Staveren e Stevenson [1982] advogam o uso da otimização peephole baseada sobre o código intermediário.

A geração de código foi tratada como um processo de reescrita de árvores por Wasilew [1971], Weingart [1973], Johnson [1978] e Cattell [1980]. O exemplo de reescrita de árvores de Seção 9.12 é derivado de Henry [1984]. Aho, Ganapathi [1985] propuseram a combinação de reescrita de árvores e técnicas de análise sintática para a geração de código. Uma outra técnica de programação dinâmica é mencionada naquela mesma seção. Lamb [1980] implementou uma linguagem de geração de código chamada Twig baseada nos esquemas de tradução de árvores da Seção 9.12. Kron [1975], Bhatt e Lee [1979] e Hoffmann e O'Donnell [1982] descrevem algoritmos gerais para o reconhecimento de padrões de árvores.

A abordagem Graham e Glanville para a geração de código utilizando uma análise sintática LR para a seleção de instruções é descrita e avaliada em Glanville [1977], Glanville e Graham [1978], Graham [1980] e [1984], Henry [1984] e Aigrain et al. [1984]. Compton [1980] e Ganapathi e Fischer [1982] usaram gramáticas de atributos para especificar e implementar geradores de código.

Outras técnicas para automatizar a construção de geradores de código foram propostas por Fraser [1977], Cattell [1980] e Leverett et al. [1980]. A portabilidade de compiladores foi tratada por Richards [1971] e [1972], Szymanski [1978] e Leverett e Szymanski [1980] descrevem técnicas para ligar instruções de desvio dependentes do deslocamento. Yuasa, Iris [1985] possui um algoritmo de tempo polinomial para o exercício 9.19.

Nota do Tradutor: ... [illegible] ... N. do T.

Capítulo 10

OTIMIZAÇÃO DE CÓDIGO

Idealmente os compiladores deveriam produzir um código-alvo que fosse tão bom quanto aquele que poderia ser escrito à mão. A realidade é que esta meta é atingida somente em casos limitados e com dificuldade. No entanto, o código produzido pelos algoritmos diretos de compilação pode ser transformado de forma a rodar mais rapidamente. A recuperação desta ineficiência é chamada de *otimização* e apresenta dois termos sem equívocos, porque só raramente pode ser garantido que o código-alvo seja o melhor possível. Os compiladores que aplicam transformações de melhoria de código são chamados de *compiladores otimizantes*.

A coleção deste capítulo está organizada em três seções independentes. A primeira trata das transformações de programa que melhoram o código gerado sem levar em consideração quaisquer propriedades da máquina-alvo. As otimizações dependentes da máquina, tais como a alocação de registradores e a utilização de seqüências específicas de instruções de máquina ideais da máquina-alvo, foram discutidas no Capítulo 9.

A maior complexidade em um algoritmo de otimização é conseguir-se perceber onde as partes do programa mais frequentemente executadas se encontram e concentrar esforços nessas partes. Há um consenso popular que diz que a maioria dos programas gasta noventa por cento do seu tempo executando cerca de dez por cento de seu código. Compiladores pessimistas, cientes desse fato, passam muito tempo otimizando uma pequena fração do programa, concluindo que a maior parte do tempo de execução. O estabelecimento de um perfil de comportamento do programa em relação ao seu tempo de execução usando-se amostras significativas de dados identifica de forma acurada as regiões pesadamente executadas pelo mesmo. Infelizmente, um compilador não pode se beneficiar de dados de entrada que serão dados a esse código, dessa forma, precisa fazer o seu melhor palpite a respeito de quais são os pontos críticos do programa.

Na prática, os laços mais internos de programas são bons candidatos para os melhoramentos. Numa linguagem que enfatize as construções de controle, tais como os comandos while e for, os laços podem ser evidentes a partir da sintaxe do programa. Em geral, um processo demorado de análise de fluxo de controle identifica os laços de grande fluxo de um programa.

Este capítulo é uma coletânea de transformações otimizantes úteis e de técnicas para implementá-las. A melhor técnica para decidir que transformações valem a pena se aplicar a uma dada parte compilada é coletar estatísticas a respeito dos programas. Iremos avaliar o benefício de um dado conjunto de otimizações em uma amostra representativa de programas-fonte reais. O Capítulo 12 descreve as transformações que foram comprovadamente úteis de serem usadas em compiladores existentes para várias linguagens diferentes.

Um dos temas deste capítulo é a análise de fluxo de dados, um processo de coletar informações a respeito das formas pelas quais as variáveis são usadas no programa. As informações coletadas nos vários pontos de um programa podem ser relacionadas usando-se uma estrutura simples de equações. Apresentamos vários algoritmos para a coleta de informações usando a análise de fluxo de dados bem como para usar essas informações na otimização. Também consideramos o impacto na otimização das construções de linguagens tais como procedimentos e apontadores.

As três primeiras seções deste capítulo falam com material mais avançado. Cobrem algumas idéias da teoria dos grafos, relevantes à análise de fluxo de controle e à aplicação da análise de fluxo de dados. O capítulo conclui com uma discussão das ferramentas de projeto para a criação de fluxo de dados e de outras para depurar o código otimizado. A ênfase ao longo deste capítulo está nas técnicas de otimização que se aplicam às linguagens em geral. Alguns compiladores que usam essas idéias serão revistos no Capítulo 12.

10.1 INTRODUÇÃO

Para criar um programa eficiente numa linguagem de alto nível, o programador precisa estar ciente de muitas coisas. Nesta seção, nós examinamos algumas opções disponíveis a um programador e a um compilador para criar programas-alvo eficientes. Mencionamos algumas classes de transformações de melhoria de código que um programador e um escritor de compiladores poderiam esperar usar de forma apropriada, independente de um programa. Consideramos também as representações de programa nas quais as transformações podem ser aplicadas.

Critérios para as Transformações de Melhoria de Código

Em resumo de forma sintética, as melhores transformações de programa são aquelas que possibilitam uma melhoria substancial com pouco esforço. As transformações providenciadas por um compilador otimizante devem ser poderosas em suas propriedades.

Primeiro, uma transformação precisa preservar o significado dos programas. Isto é, uma "otimização" não pode modificar a saída pro-

[1] Em certas circunstâncias, como em uma rotina de biblioteca, pode-se desejar um código-alvo que seja o mais rápido possível, independentemente do tempo de compilação ou do tamanho do código-alvo. Mais adiante, no Capítulo 9, discutimos a geração de código ótimo, que é caro em termos de tempo de compilação e de espaço.

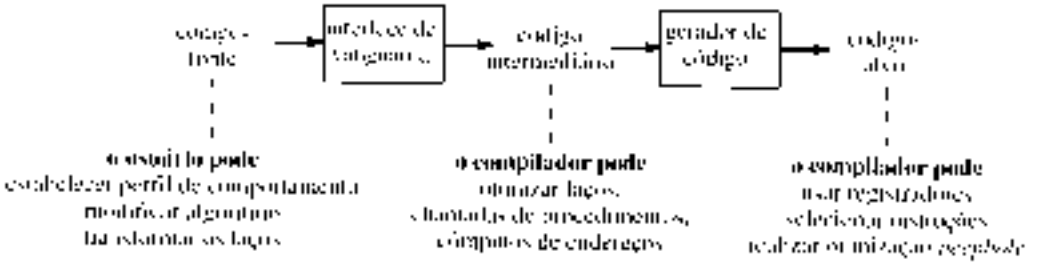


Fig. 10.1. Locais para melhorias (usuário e compilador) durante a compilação

dizendo por um procedimento para uma dada entrada (ou caso) em especial. Como uma alternativa, poderíamos pensar em uma versão original do programa-fonte. A influência deste critério se espalha por este capítulo, com todas as vezes sabiamente o critério "original" de perder uma oportunidade de aplicar uma transformação em vez de correr o risco de "tornar esperto" o programa.

Segundo, uma transformação precisa ser benéfica. Excelentes programas "não são construídos" imediatamente. Algumas vezes, estamos interessados em reduzir o espaço usado pelo código compilado, apesar de, mais frequentemente, querermos uma melhoria de velocidade. Naturalmente, nem toda transformação terá sucesso em melhorar qualquer objetivo específico. Ocasionalmente, uma "otimização" pode ser danosa ligeiramente em um programa, fazendo com que a versão melhorada esteja de certa forma em desvantagem.

Terceiro, uma transformação precisa valer a pena. Não faz sentido que um escritor de compiladores dispenda esforço intelectual para implementar uma transformação de melhoria de código, e então o compilador gaste tempo adicional de compilação com o programa. Mais ainda, se esse esforço não for recompensado quando o programa-alvo for executado. Certas transformações locais (*peephole*, do tipo discutido na Seção 9.9, são simples e poderosas o suficiente para serem incluídas em qualquer compilador.

Algumas transformações podem somente ser aplicadas após uma análise detalhada (frequentemente denominada de programa-fonte). Desta forma, existe pouca discussão sobre se devem ou não ser aplicadas a programas que serão usados por apenas uma pequena vez. Por exemplo, um compilador rápido, não otimizante, está projetado para ser muito similar a um depurador ou quando usado em "aplicações de estudantes", que serão executadas com sucesso em algumas poucas vezes e abandonadas. Somente quando o programa em questão ocupa uma fração significativa dos ciclos de tempo de CPU que a qualidade do código otimizado justifica o tempo gasto pelo compilador otimizante para o mesmo.

Obtendo Melhor Desempenho

Melhorias dramáticas no tempo de execução de um programa — tais como a redução do tempo de execução de horas para minutos ou segundos — são usualmente obtidas melhorando-se o programa em todos os níveis, a partir do nível mais alto de projeto, como sugerido pela Fig. 10.1. A cada nível, as oportunidades diminuem em escala entre os dois extremos de se encontrar um melhor algoritmo ou implementar um dado algoritmo específico de forma que um número menor de operações seja realizado.

As transformações algorítmicas produzem ocasionalmente melhorias espetaculares no tempo de execução. Por exemplo, Bentley [1982] relata que o tempo de execução de um programa para classificar N elementos caiu de $2{,}02N^2$ microssegundos para $12N\log N$ microssegundos, quando uma "classificação de inserção" relativamente sofisticada foi substituída por um "quicksort". Para $N = 1000$, a substituição acelerou o programa por um fator de 2,5. Para $N = 1000000$, a melhoria é substancial: observe diretamente acelerou o programa por um fator de mais de 200.

Infelizmente, nenhum compilador pode encontrar a melhor algoritmo para um dado programa. Algumas vezes, entretanto, um compilador pode substituir uma seqüência de operações por uma outra, gerando diretamente um código que execute mais rapidamente. Reduzir significativamente o tempo de execução de um programa. Estes saltos são uma das quais são aplicadas transformações algébricas a programas em linguagens em nível mais alto, como, por exemplo, as linguagens de interrogação a bancos de dados (ver Ullman [1982]).

Nesta seção e na próxima, um programa de classificação chamado Quicksort será usado para ilustrar o efeito das várias transformações de melhoria de código. O programa C da Fig. 10.2 é derivado de Sedgewick [1978], onde uma otimização manual do mesmo é discutida. Nos concentraremos aqui nos aspectos algorítmicos do programa — de fato, [illegible] precisa ter o menor elemento [illegible] e [illegible] e menor, para que o programa funcione.

Pode não ser possível realizar certas transformações de melhoria no código no nível da linguagem-fonte. Por exemplo, uma linguagem como Pascal ou Fortran, o programador pode somente se referir a elementos de um vetor da forma usual, como, por exemplo, [illegible]. Ao nível da linguagem intermediária, entretanto, novas oportunidades para melhoria de código podem ser expostas. O código de três endereços, por exemplo, fornece muitas oportunidades para a melhoria do cômputo de endereços, especialmente dentro de laços. Correspondentemente, o código de três endereços para obter um valor de [illegible] mostra que cada elemento do vetor ocupa quatro bytes.

[illegible]

[illegible]

Fig. 10.2. Código C para quicksort.

[illegible] Ver Abelson, Sussman e Sussman [1985] para uma discussão sobre como [illegible] dados.

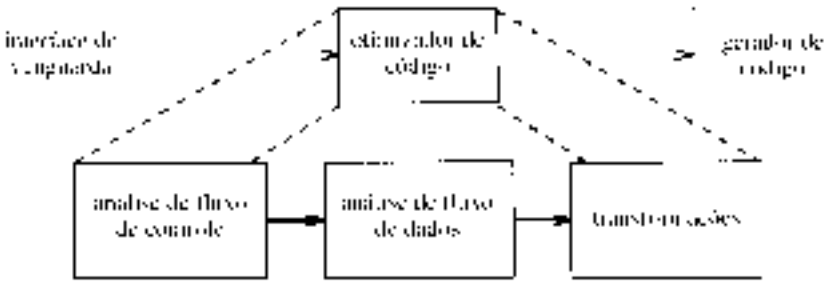


Fig. 10.3. Organização típica de um otimizador de código.

Um código ingênuo de três endereços para calcular [illegible] cada vez que [illegible] ligações para um otimizador de programa não possam remover todos os cômputos redundantes de endereços mais complexos ou aplicações de aritmética de ponteiros, em vez de serem explícitos no código escrito pelo programador. Nessas situações, é muito útil ao compilador ter a capacidade de eliminar cômputos redundantes. Muitas linguagens como C permitem que essa transformação seja feita ao nível da linguagem fonte, uma vez que os elementos de arranjos podem ser sistematicamente referenciados usando apontadores para locais específicos. Essa abordagem é mais trabalhosa para as implementações que os compiladores otimizantes de bom nível podem aplicar.

Assim, a tarefa mais difícil e importante de responsabilidade do compilador é fazer o melhor uso de seus recursos. Por exemplo, as máquinas com registradores pessoais são muito mais rápidas se os registradores puderem ser usados eficientemente. Frequentemente, o tempo de execução é dominado pelo tempo que leva para mover certas variáveis para dentro e para fora dos registradores, mas um bom compilador pode fazer um trabalho muito melhor do que um programador ingênuo. Semelhantemente, uma máquina com pipeline pode executar significativamente mais rápido os programas que evitam desvios condicionais, pois os custos de interrupção do pipeline são evitados. Uma máquina com múltiplos processadores pode executar muito mais rápido os programas que explorem o paralelismo inerente a certos algoritmos. Infelizmente, o projeto de algoritmos que explorem eficientemente máquinas paralelas é uma área de pesquisa ativa, e não existem ainda técnicas de compilação bem compreendidas para a maioria das máquinas paralelas. Os compiladores para máquinas paralelas são discutidos no Capítulo 9.

Mesmo que seja possível para o programador melhorar o código gerado por um compilador, é preferível deixar o compilador realizar algumas das melhorias. Se a mesma compilação pode ser executada a geração de um código eficiente, o usuário pode se concentrar em escrever um código correto e

Uma Organização para um Compilador Otimizante

Como mencionamos, existem frequentemente muitas oportunidades nas quais um programa pode ser melhorado. Como as técnicas para analisar e transformar um programa são muito significativamente complexas, este capítulo se concentra nas transformações do código intermediário usando a organização mostrada na Fig. 10.3. A fase de melhoramento do código consiste nas análises de fluxo de controle e de fluxo de dados, seguidas pela aplicação de transformações. O gerador de código discutido no Capítulo 9 produz um programa-alvo a partir do código intermediário transformado.

Por uma questão de conveniência da apresentação, assumimos que o código intermediário consista em um código de três endereços. O código intermediário do tipo produzido pelas técnicas do Capítulo 8 para a parte do programa da Fig. 10.2 é mostrado na Fig. 10.4. Como outras representações intermediárias, as variáveis temporárias [illegible] não precisam ligar explicitamente, como discutido no Capítulo 8.

A organização da Fig. 10.3 possui as seguintes vantagens:

1. As operações necessárias para implementar as construções de alto nível são tornadas explícitas no código intermediário, de modo que possa ser otimizado. Por exemplo, os cômputos de endereços para [illegible] são explícitos na Fig. 10.4, de modo que expressões comuns [illegible] podem ser eliminadas, como discutido na próxima seção.

2. O código intermediário pode ser relativamente independente da máquina-alvo, de modo que o otimizador e o gerador de código possam ser substituídos por outros de uma máquina diferente. O código intermediário da Fig. 10.4 assume que cada elemento de um array [illegible] ocupa quatro bytes. Alguns códigos independentes, como, por exemplo, o código P para Pascal, devem ser usados para gerador de código preencher os tamanhos dos elementos de arranjos, de forma que o código intermediário seja independente do tamanho da palavra de máquina. Poderíamos ter feito o mesmo em nosso código intermediário, substituindo os 4 por uma constante simbólica.

```
(1)   i := m-1
(2)   j := n
(3)   t1 := 4*n
(4)   v := a[t1]
(5)   i := i+1
(6)   t2 := 4*i
(7)   t3 := a[t2]
(8)   if t3 < v goto (5)
(9)   j := j-1
(10)  t4 := 4*j
(11)  t5 := a[t4]
(12)  if t5 > v goto (9)
(13)  if i >= j goto (23)
(14)  t6 := 4*i
(15)  x := a[t6]
(16)  t7 := 4*i
(17)  t8 := 4*j
(18)  t9 := a[t8]
(19)  a[t7] := t9
(20)  t10 := 4*j
(21)  a[t10] := x
(22)  goto (5)
(23)  t11 := 4*i
(24)  x := a[t11]
(25)  t12 := 4*i
(26)  t13 := 4*n
(27)  t14 := a[t13]
(28)  a[t12] := t14
(29)  t15 := 4*n
(30)  a[t15] := x
```

Fig. 10.4. Código de três endereços para o fragmento de código da Fig. 10.2.

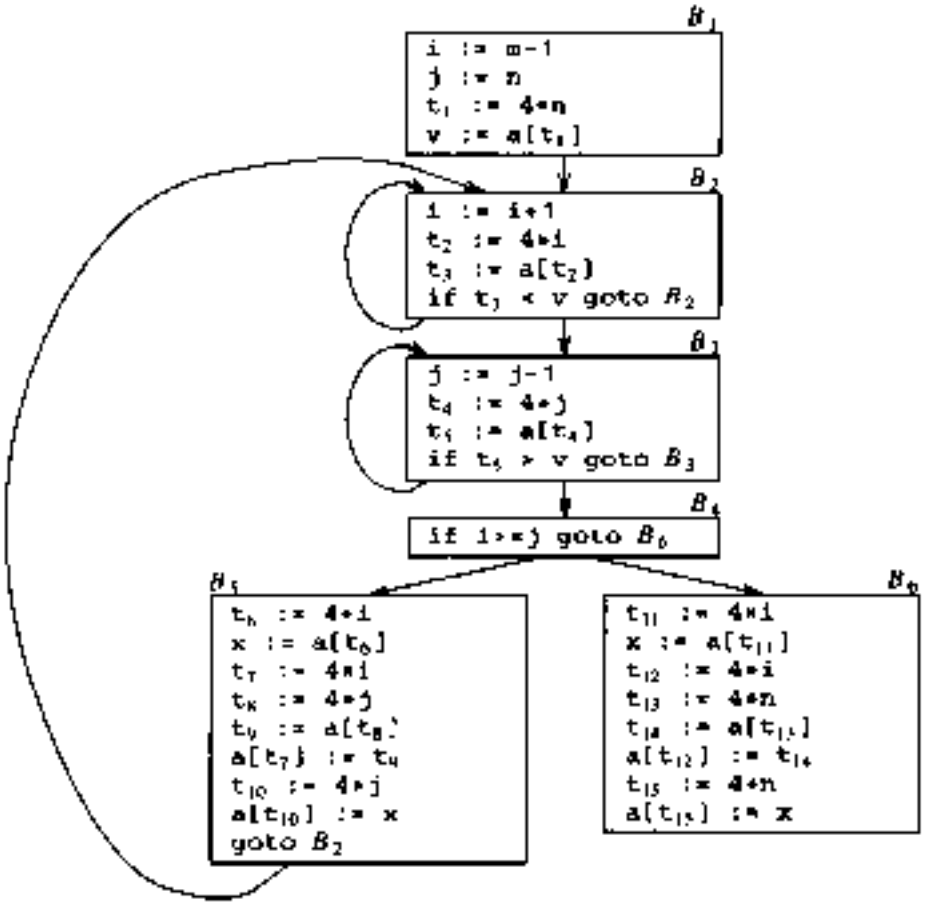


Fig. 10.5. Grafo de fluxo.

No otimizador de código, os programas são representados por grafos de fluxo, nos quais os nós representam blocos básicos, como discutido na Seção 9.1. A menos que o contrário seja dito, estamos assumindo que o programa opera em um nível procedimental. Na Seção 10.8, discutiremos a otimização interprocedimental.

Exemplo 10.1. A Fig. 10.5 contém o grafo de fluxo para o programa da Fig. 10.4. B_1 é o nó inicial. Todos os desvios condicionais e incondicionais para os enumerados na Fig. 10.4 foram substituídos, na Fig. 10.5, por desvios para os blocos nos quais os comandos destinos são os líderes.

Na Fig. 10.5 existem três laços, B_2 e B_3 são laços por si mesmos. Os blocos B_2, B_3 e B_4 formam, juntos, um laço com entrada em B_2. □

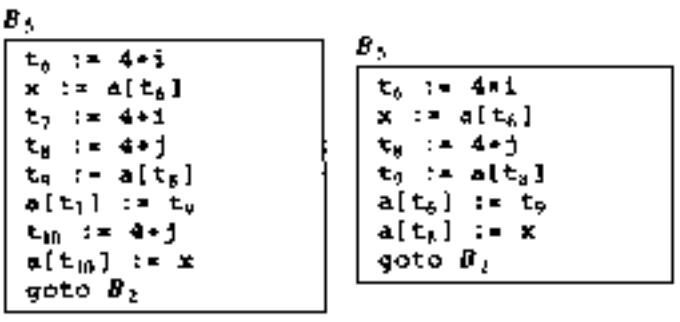
10.2 AS PRINCIPAIS FONTES DE OTIMIZAÇÃO

Nesta seção, introduziremos algumas das mais importantes transformações para melhoria de código. As técnicas para implementar essas transformações são apresentadas nas seções subsequentes. Uma transformação de programa é chamada *local* se ela puder ser realizada examinando-se apenas os comandos básicos que constituem o bloco básico correspondente, e *global* se for feita sobre todo o grafo de fluxo. Muitas transformações podem ser realizadas em ambos os níveis, local e global. As transformações locais são geralmente realizadas primeiro.

Transformações Função-Preservantes

Existem várias formas através das quais um compilador pode melhorar um programa sem modificar a função que o mesmo computa. A eliminação de subexpressões comuns, a propagação de cópias, a eliminação de código morto e a transposição para constantes são exemplos comuns de tais transformações função-preservantes. A Seção 9.8 sobre a representação de blocos básicos sob a forma de GDAs mostrou como as subexpressões comuns locais podem ser detectadas e removidas pela construção do GDA para o bloco básico. Nesse sentido construído. As outras transformações surgem primariamente quando as otimizações globais são feitas, e discutiremos uma de cada vez.

Um programador pode incluir frequentemente vários cálculos do mesmo valor, tais como o deslocamento de um array. Como mencionado na Seção 10.1, alguns desses cálculos não podem ser evitados pelo programador, porque ficam abaixo do nível de detalhe acessível dentro da linguagem-fonte. Por exemplo, o bloco B_5 mostrado na Fig. 10.6(a) recalcula $4*i$ e $4*j$.



(a) Antes (b) Depois

Fig. 10.6. Eliminação de subexpressões comuns locais.

Subexpressões Comuns

Uma ocorrência de uma expressão E é chamada de uma *subexpressão comum* se E foi computada previamente e os valores das variáveis em E não mudaram desde a ocorrência anterior. Podemos evitar recomputar uma subexpressão se pudermos usar o valor previamente calculado. Por exemplo, as atribuições $t_7 := 4*i$ e $t_9 := 4*i$ apresentam subexpressões comuns $4*i$ e $4*j$, respectivamente, no código do meio da Fig. 10.6(a). Depois de implementadas na Fig. 10.6(b), usando-se t_7 em lugar de t_9 e t_6 em lugar de t_{10}, essa mudança é o que irá resultar se otimizarmos o código intermediário a partir do DAG para o bloco básico.

Exemplo 10.2. A Fig. 10.7 mostra o resultado de se eliminarem as subexpressões comuns – tanto locais quanto globais – dos blocos B_3 e B_4 no grafo de fluxo da Fig. 10.5. Discutiremos primeiro a transformação de B_3 e, em seguida, mencionaremos algumas das sutilezas que envolvem os arrays.

Depois que as subexpressões locais comuns foram eliminadas, B_3 só irá recalcular $4*i$ e $4*j$, como mostrado na Fig. 10.6(b). Ambas são subexpressões comuns, em particular, as três encontradas

em B_3, podem ser substituídas por

pois t_2 é computado no bloco B_2. Na Fig. 10.7, observe-se que, à medida que o controle passa da atribuição em B_2 para B_3, não existe qualquer atribuição em t_2, dessa forma t_2 pode ser usado quando $4*i$ for necessário.

Uma sutil subexpressão comum surge aqui em B_3 após a substituição: $a[t_4]$. Ela é a expressão $a[t_6]$, correspondente, ao nível de código fonte, ao valor de $a[j]$. Mais sutilmente, $a[t_6]$ pode ter o valor quando o controle abandona B_3, se em seguida entra em B_3, mas também

que é o valor computado por $a[t_4]$. Dessa forma, a posição ocupada por atribuições a elementos do array a pode mudar. Os comandos

em B_3 podem, por conseguinte, ser substituídos por

Analogamente, o valor atribuído a x no bloco B_3 da Fig. 10.6(b) é visto ser a mesma que o valor atribuído a x no bloco B_5. O bloco B_6 na Fig. 10.7, o resultado da eliminação das subexpressões comuns correspondentes aos valores das expressões em nível fonte $a[j]$ e $a[i]$, de B_6 na Fig. 10.6(b), é uma versão similar de transformações feitas em B_5, só que existem atribuições a a. Por conseguinte, $a[t_6]$ pode não ter o mesmo valor ao se atingir B_6, como o era ao deixar B_5, e não é seguro tratar $a[t_6]$ como uma subexpressão comum.

Propagação de Cópias

O bloco B_5 na Fig. 10.7 pode ser subsequentemente melhorado através da eliminação de x usando-se dois novos tipos de transformações. Uma delas envolve atribuições da forma $x := y$, chamadas de comandos de cópia, ou apenas cópias, abreviadamente. Em essência, as cópias não são necessárias. Por exemplo, no código da Fig. 10.8, as cópias em B_5 foram criadas porque a regra que gera código para uma atribuição de array $x[i] := y$ requer que x seja colocado em um registrador. Em B_5, após a atribuição a x, a nova x é imediatamente substituída por t_3. Dentro de B_5 (após x receber o valor de t_3), o uso de x pode ser substituído por t_3, pois nenhuma atribuição a x ocorre nesse bloco.

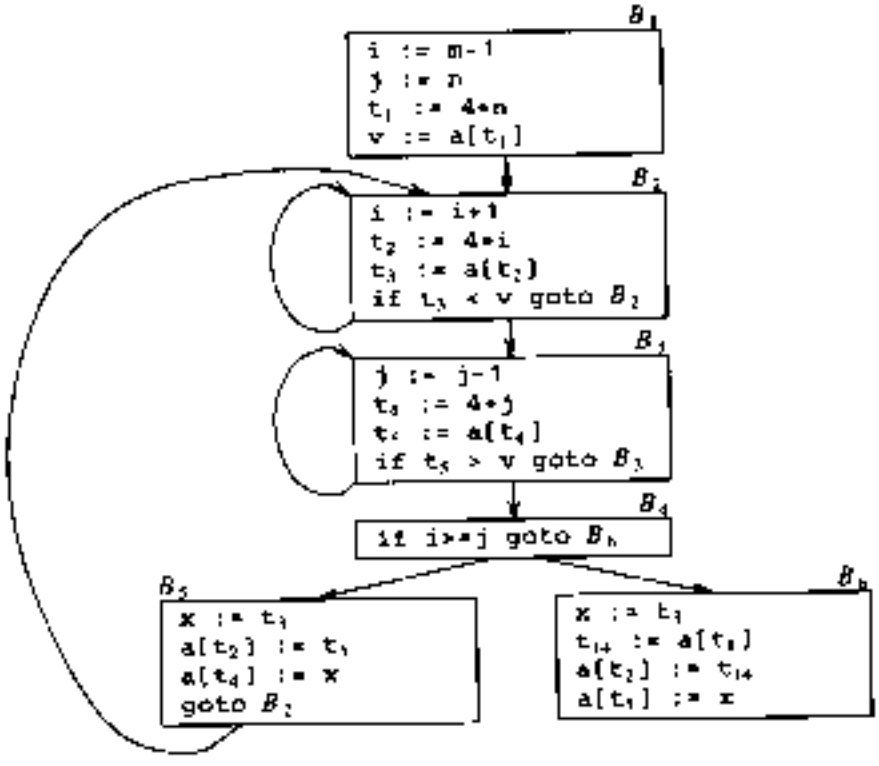


Fig. 10.7. B_3 e B_4 após a eliminação de subexpressões comuns

A idéia por trás da transformação da propagação de cópias é substituir em lugar de x se aplique que possível. Após eventuais subdivisões, por exemplo, a atribuição $x = y$ no bloco B_5 da Fig. 10.7 é uma cópia. A propagação de cópias aplicada a B_5 produz:

$$\begin{array}{l} \cdots \\ \cdots \\ \cdots \\ \cdots \\ B_5 \end{array} \qquad (10.1)$$

Isso pode não parecer uma melhoria, mas, como veremos, nos dá a oportunidade de eliminação a atribuição a x.

Eliminação de Código Morto

Uma variável está viva em um ponto do programa se seu valor puder ser usado subsequentemente; em caso contrário, está morta naquele ponto. Ocasionalmente, relacionadas a algoritmos mal projetados, ocorrências que computam um valor que jamais será usado. Embora programadores experientes provavelmente não produzam código desse tipo de forma intencional, o mesmo pode surgir como resultado de transformações prévias. Por exemplo, na Seção 10.9, discutiremos o caso de [illegible] que é estabelecida para verdadeiro em todos os caminhos para um ponto do programa, e usada em uma instrução como

$$\cdots \quad \cdots \qquad (10.2)$$

Através de uma análise de fluxo de dados, pode ser possível deduzir que, cada vez que o programa atinja esse ponto, o valor de [illegible] seja falso. Usualmente, o mais provável é que existe algum erro de [illegible] particular.

$$\cdots \quad \cdots$$

que podemos deduzir que seja a última atribuição a [illegible] antes da instrução (10.2). Não importa que sequência de otimizações o programa efetivamente siga. Se uma propagação de cópias substituir [illegible] para [illegible] e uma análise de impressão eventualmente mostrar que a atribuição [illegible] não pode ser alcançada, podemos eliminar, do código objeto, tanto o teste quanto a impressão. Mais geralmente, dentro de um programa, uma expressão que consiste de uma constante é uma constante cujo valor pode ser determinado durante a compilação, pode ser substituída por essa constante. Essa transformação é chamada *dobramento de constantes*.

Uma variante da propagação de cópias é muito frequentemente transformação causada pelas próprias cópias. Por exemplo, após a propagação de cópias, seguida pela eliminação de código morto, a remoção da atribuição $x = y$ resulta em:

$$\begin{array}{l} \cdots \\ \vdots \\ B_5 \end{array}$$

Esse código representa uma melhoria subsequente do bloco B_5 na Fig. 10.2.

Otimizações de Laços

Concentramos a nossa breve introdução a otimização de código em duas importantes transformações, nomeadamente os laços, e especialmente os mais internos, onde os programas tendem a passar a maior parte de seus tempos. O tempo de execução de um programa pode ser melhorado se decrementarmos o número de instruções de um laço interno, mesmo que aumentemos o número de instruções de um laço mais externo. Frequentemente, podemos obter muito código fora de um laço mesmo. Três técnicas são importantes para a otimização de laços: a movimentação de código, que move código de um laço; a eliminação de variáveis de indução, que altera [illegible] para [illegible] em um laço; e a redução de força, que substitui uma operação por uma outra mais barata em alguns casos, tal como uma multiplicação por uma adição.

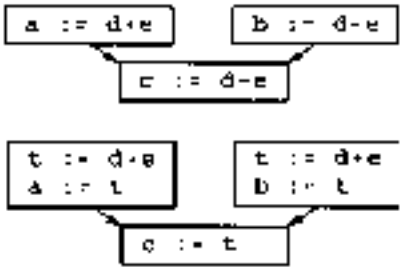


Fig. 10.8. Cópias introduzidas durante a eliminação de subexpressões comuns.

Movimentação de Código

Uma importante melhoria que diminui a quantidade de código em um laço é a movimentação de código. Esta transformação toma um cálculo que produz o mesmo resultado independentemente de quantas vezes que o laço é executado, *uma expressão invariante do laço*, e coloca-o antes do mesmo. Note-se que a noção "antes do laço" assume a existência de uma entrada para o laço. Por exemplo, a avaliação de [illegible] é invariante nos seguintes [illegible]:

[illegible] é calculada uma vez

A movimentação de código invariante resulta no código equivalente a:

[illegible] é calculada uma vez

Variáveis de Indução e Redução de Capacidade

Conquanto a movimentação de código não se aplique a um exemplo dado, quase todos os laços contêm variáveis de indução. Muitas outras transformações são. Os laços são processados de dentro para fora. Por exemplo, consideramos o laço em torno de B_3. Somente a parte do grafo de fluxo e [illegible] relevantes às transformações sobre B_3 é mostrada na Fig. 10.9.

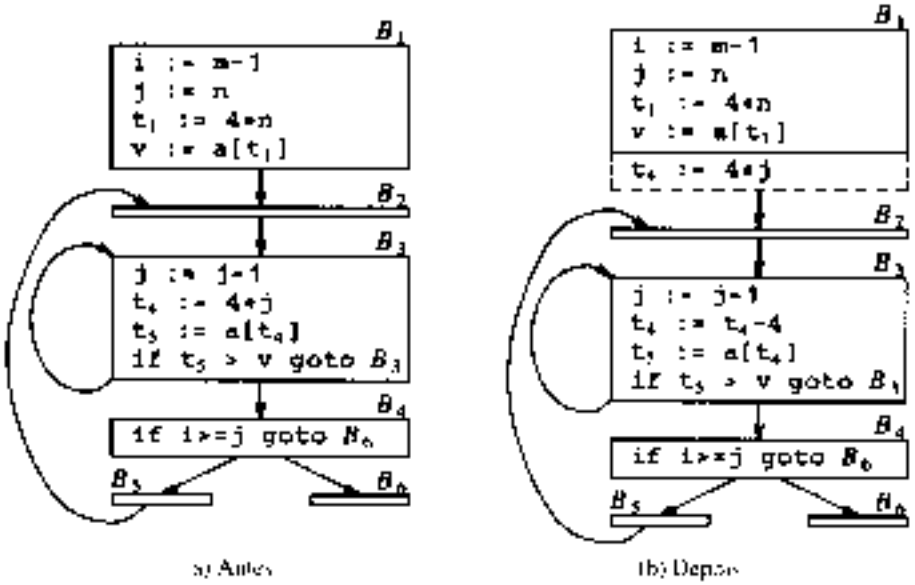
Note-se que os valores de [illegible] permanecem para passar a cada vez que o valor de [illegible] decresce de 1. Quando [illegible] decresce de 1 para [illegible], [illegible] é atribuído a [illegible]. Tais valores são chamados de *variáveis de indução*.

Quando existem duas ou mais variáveis de indução em um laço, pode ser possível eliminar todas menos uma, durante o processo de eliminação das variáveis de indução. Para o laço mais interno em torno de B_3 na Fig. 10.9(c), o teste [illegible] pode ser substituído completamente, [illegible] usada em B_3 e em B_4. No entanto, podemos desenvolver uma parte do processo de eliminação de variáveis de indução sequencialmente. [illegible] será eliminada quando o laço mais externo de B_3-B_4 for considerado.

Exemplo 10.3. Na medida em que o relacionamento [illegible] é certamente válido após uma tal atribuição a [illegible] na Fig. 10.9(a), [illegible] pode ser usado em qualquer outro ponto no laço mais interno, ao longe de B_3, sempre que [illegible] estiver após [illegible] e não houver [illegible] a seleção [illegible].

Deverá que ser [illegible]. Podemos, então, substituir a atribuição [illegible] por [illegible]. O único problema restante é que [illegible] não tem um valor inicial correto no bloco B_1, pela primeira vez. Como precisamos manter o relacionamento [illegible] sempre que [illegible] for atribuído, [illegible] é inicializado, imediatamente antes do bloco B_2, na Fig. 10.9(b).

Para todos os efeitos práticos, [illegible]



a) Antes b) Depois

Fig. 10.9. Redução de capacidade aplicada a *t4* no bloco B_3.

A substituição de uma multiplicação por uma subtração irá acelerar o código objeto se a multiplicação levar mais tempo do que a adição ou subtração, como é o caso de muitas máquinas.

A Seção 10.7 discute como as variáveis de indução podem ser detectadas e que transformações podem ser aplicadas. O restante desta seção contém mais um exemplo de eliminação de variáveis de indução, o qual também envolve o reconhecimento de laços mais externos contendo B_3. B_4 e B_5.

Exemplo 10.4. Após a redução de capacidade ser aplicada aos laços que têm apenas um bloco de B_3 e B_5, é conveniente determinar o resultado do teste no bloco B_4. Sabemos que os valores de *t2* e de *t4* satisfazem a relação *t2* = *t4* + 4, enquanto que aqueles de *i* e *j* satisfazem a relação *j* <= *i*. De forma que o teste *t2* >= *t4* é equivalente a *t4* >= *t4*. Uma vez que essa substituição seja feita, *t2* no bloco B_3 e em B_5 se tornam variáveis mortas e as atribuições a elas nesses blocos se tornam código morto, o qual pode ser eliminado, resultando no grafo de fluxo mostrado na Fig. 10.10. □

As transformações de melhoria de código foram aplicadas. Na Fig. 10.10, o número de instruções nos blocos B_3 e B_5 reduziu-se de 4 para 3 em cada um, a partir do grafo de fluxo original na Fig. 10.5 em B_3, reduziu-se de 9 para 3 e em B_5, de 8 para 3. Por outro lado, B_2 cresceu de quatro para seis instruções, mas o mesmo só é executado uma vez no fragmento de código, de forma que o tempo total de execução é só levemente afetado pelo tamanho de B_2.

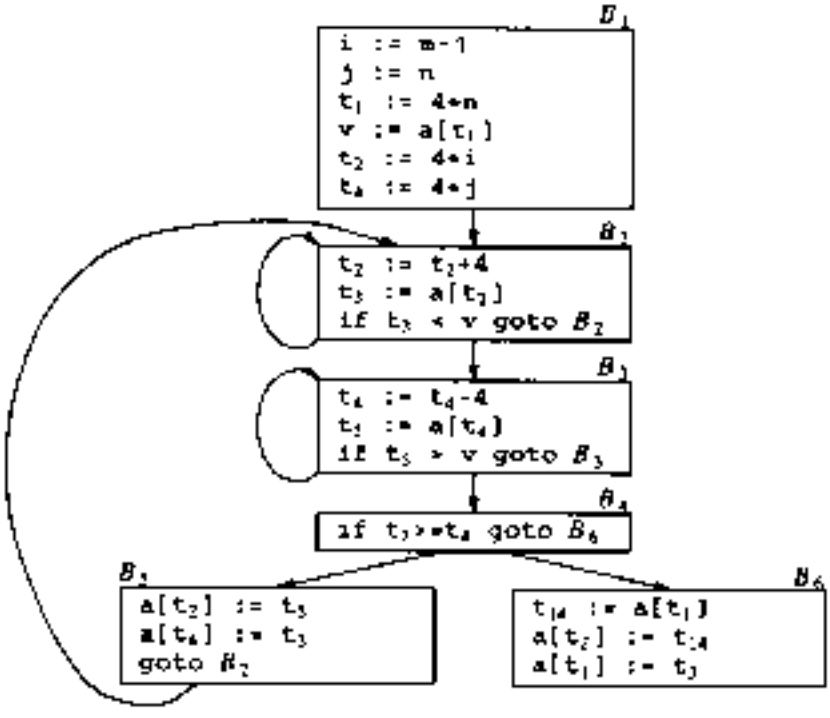


Fig. 10.10. Grafo de fluxo após a eliminação de variáveis de indução.

10.3 OTIMIZAÇÃO DOS BLOCOS BÁSICOS

No Capítulo 9, examinamos várias transformações de preservação de código para os blocos básicos. Incluíam transformações estrutura-preservadoras, tais como a eliminação de código morto e as transformações algébricas, tais como a redução de força.

Muitas das transformações estrutura-preservadoras podem ser implementadas através da construção de um GDA para um bloco básico. Relembremos que existe um nó no GDA para cada um dos valores iniciais das variáveis que aparecem no bloco básico e que existe um nó associado a cada enunciado dentro do bloco. Os filhos desse nó são aqueles nós que correspondem aos enunciados que são as últimas definições, dentro do bloco, dos operandos usados pelo enunciado. O nó é rotulado pelo operador aplicado a ele, também acrescida a ele, está a lista de variáveis para as quais esta é a última definição dentro do bloco. Também anotamos aqueles nós, se algum, cujos valores estão previstos para saída do bloco; esses são os nós de saída.

As subexpressões comuns podem ser detectadas sempre que, ao tentarmos criar um novo nó a ser rotulado com o operador op e com os mesmos filhos, na mesma ordem, l e r, encontramos um nó existente com os mesmos filhos, na mesma ordem, e com o mesmo operador. Se assim for, l já computa o mesmo valor que $op(l, r)$ poderia ser usado em seu lugar.

Exemplo 10.5. Um GDA para o bloco (10.2):

$$\begin{aligned} a &= b - c \\ b &= b - d \\ c &= a + c \\ d &= a \end{aligned} \tag{10.2}$$

é mostrado na Fig. 10.11. Quando estivermos construindo o nó para o terceiro enunciado, $c = a + c$, sabemos que o uso de a se refere ao nó da Fig. 10.11 rotulado $-$, porque esse é a definição mais recente de b. Por conseguinte, não confundimos os valores computados nos enunciados um e três.

No entanto, o nó correspondente ao quarto enunciado, $a = c$, tem $+$ para o operador e os mesmos dois filhos, a_0 e c_0, como antes. Uma vez que o operador e os filhos são os mesmos que aqueles para o nó correspondente ao enunciado três, não criamos este nó, mas adicionamos a à lista de definições para o nó rotulado $+$.

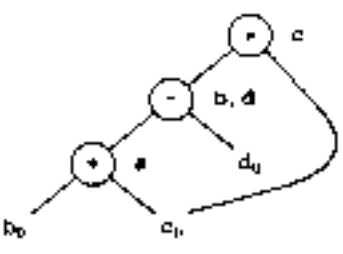


Fig. 10.11. Um GDA para o bloco básico (10.2).

Poderia parecer que, como existem somente três nós no GDA da Fig. 10.11, o bloco (10.3) poderia ser substituído por um bloco com somente três enunciados. De fato, a ou c (uma das duas) estaria viva à saída do bloco, não precisaríamos computar aquela variável e poderíamos usar a outra para receber o valor representado pelo nó rotulado $+$ na Fig. 10.11. Por exemplo, se b não estivesse viva à saída do bloco, poderíamos usar

$$\begin{aligned} a &= b - c \\ c &= a + c \\ a &= c \end{aligned}$$

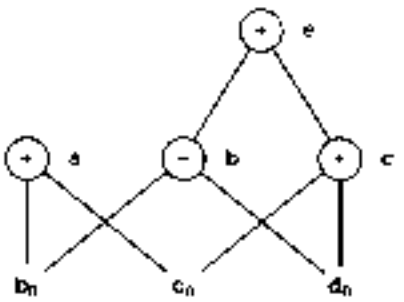


Fig. 10.12. GDA para o bloco básico (10.4).

No entanto, se ambos, b e c, estiverem vivos à saída, um quarto enunciado precisa ser usado para copiar o valor de uma para outra.[1]

Note-se que, quando procuramos por subexpressões comuns, estamos realmente procurando por expressões que garantam computar o mesmo valor, não importa como o valor seja computado. Por conseguinte, o método do GDA na verdade faz de que a expressão computada pelos primeiro e quarto enunciados na sequência

$$\begin{aligned} a &= b - c \\ c &= a + c \\ a &= b - c \\ d &= a \end{aligned} \tag{10.4}$$

é a mesma, nomeadamente $b_0 - c_0$. No entanto, as identidades algébricas aplicadas ao GDA, como discutido em seguida, podem expor a equivalência. O GDA para esta sequência é mostrado na Fig. 10.12.

A operação sobre os GDAs que corresponde à eliminação de código morto é claramente direta de se implementar. Removemos do GDA qualquer raiz (nó sem ancestrais) que não tenha variáveis vivas. A repetida aplicação desta transformação irá remover todos os nós do GDA que correspondam a código morto.

O Uso de Identidades Algébricas

As identidades algébricas representam uma outra classe importante de otimizações sobre os blocos básicos. Na Seção 9.1, introduzimos algumas transformações algébricas simples que poderiam ser usadas durante uma otimização. Por exemplo, podemos aplicar identidades aritméticas, tais como

$$\begin{aligned} x &= x + 0 \quad \text{ou} \quad x = 0 + x \quad \text{ou} \quad x = x - 0 \quad \text{ou} \quad x = x \\ x &= x * 1 \\ x &= 1 * x \quad \text{ou} \quad x = x / 1 \\ x &= x \end{aligned}$$

Outra classe de otimizações algébricas inclui a redução de capacidade, isto é, a substituição de um operador mais dispendioso por um mais econômico, como em

$$\begin{aligned} x &= x * 2 \quad \text{ou} \quad x = 2 * x \quad \text{ou} \quad x = x + x \\ x &= x / 2 \quad \text{ou} \quad x = x * 0.5 \\ x &= x * 8 \quad \text{ou} \quad x = 8 * x \end{aligned}$$

Uma terceira classe de otimizações relacionadas é a transposição (ou constantes em tempo

[1] Por outro lado, se ambos b e c estiverem vivos à saída, ainda podemos usar o GDA, desde que estejamos cientes de que, em tempo de execução, um dos valores corresponde aos valores de b e o outro aos valores de c. Se uma ou outra variável for definida duas vezes, como foi feito em nosso exemplo, também podemos precisar de um enunciado extra para copiar um valor de uma variável para a outra, se for necessário preservar ambos os valores corretamente.

de compilação e as substituímos pelos seus valores. Por conseguinte, a expressão [illegible] será substituída por [illegible]. Muitas expressões constantes emergem durante o uso de constantes simbólicas.

O processo de construção de GDAs pode nos auxiliar a descobrir e empregar transformações algébricas para gerar códigos de objeto eficientes e de uso geral. Por exemplo, suponhamos que [illegible] seja comutativo, [illegible]. Antes de criar um novo nó para [illegible], o compilador procurará um nó rotulado [illegible]. Procuraremos, ainda, por um nó rotulado [illegible]. Dependendo da natureza de [illegible], um ou nenhum desses nós poderá existir.

Os operadores relacionais [illegible] e, em alguns casos, para subexpressões comuns inesperadas. Por exemplo, a condição [illegible] pode também ser testada pela subtração de argumentos e verificando se um teste do código de condição estabelecido pela subtração. A subtração pode, entretanto, introduzir erros de capacidade de representação de valor quando os representações não são representáveis ou representáveis a uma dada precisão. *Entretanto*, quando a otimização de computação não é ativada, [illegible] será gerado para [illegible].

As leis associativas também podem ser aplicadas de forma a expor subexpressões comuns. Por exemplo, se o código-fonte possui as atribuições

[illegible]

o seguinte código intermediário poderia ser gerado:

[illegible]

Se [illegible] for [illegible] fora do bloco, [illegible] a sequência para

[illegible]

instrui a inserção de [illegible] em vez de [illegible].

O escritor de compiladores deve examinar cuidadosamente a especificação da linguagem, de forma a determinar que reorganizações das computações são permitidas. Uma vez que a aritmética de computadores nem sempre obedece às identidades algébricas da matemática. Por exemplo, o padrão para Fortran 77 estabelece que um compilador pode alterar qualquer expressão matematicamente equivalente, porém desde que [illegible] e os parênteses não sejam violados. Por conseguinte, um compilador pode avaliar [illegible] como [illegible], mas não pode avaliar [illegible] como [illegible]. O compilador Fortran precisa, consequentemente, controlar onde os parênteses estavam presentes nas expressões de linguagem-fonte. Isso dificulta a otimização de programas de acordo com a definição da linguagem.

10.4 LAÇOS EM GRAFOS DE FLUXO

Antes de considerarmos otimizações de laços, precisamos definir um conceito de laço. Usaremos a noção de um nó "dominando" um outro para definirmos "laços naturais" e uma importante classe essencial de grafos de fluxo "reduzíveis". Um algoritmo para se encontrarem os dominadores e verificar a redutibilidade de grafos de fluxo será fornecido na Seção 10.9.

Se [illegible] for [illegible] e [illegible] for [illegible], [illegible] será [illegible]. Dependendo [illegible] e [illegible], [illegible] pode ser [illegible] ou [illegible]. Se [illegible] for [illegible], [illegible] será [illegible]. Se [illegible] for [illegible], [illegible] será [illegible].

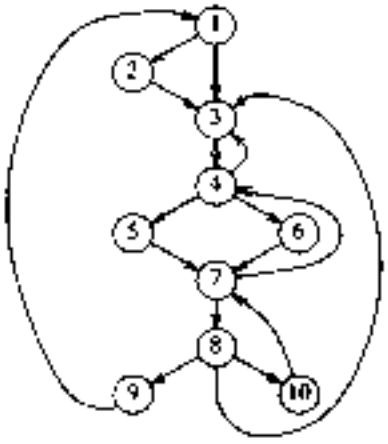


Fig. 10.13. Grafo de fluxo.

Dominadores

Dizemos que um nó d de um grafo de fluxo *domina* um nó n, escrito *d* domina *n*, se todo caminho percorrido a partir do nó inicial do grafo até n passar através de d. Sob esta definição, cada nó domina a si mesmo e o nó inicial domina todos os nós alcançáveis. Outra definição, também usada na Seção 9.4, é dominar todos os nós do laço.

Exemplo 10.6. Consideremos o grafo de fluxo da Fig. 10.13, com nó inicial 1. O nó inicial domina a si mesmo. O nó 2 domina somente a si mesmo, uma vez que o controle pode atingir qualquer outro nó ao longo de um percurso que comece por $1 \rightarrow 3$. O nó 3 domina todos os nós, exceto os nós 1 e 2. O nó 4 domina todos os nós, exceto 1, 2 e 3, já que todos os percursos a partir de 1 precisam começar $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ ou $1 \rightarrow 3 \rightarrow 4$. Os nós 5 e 6 dominam somente a si mesmos, uma vez que o fluxo de controle pode evitar alternativamente um ou outro, bem como. 7 domina 7, 8, 9, 10; 8 domina 8, 9, 10; 9 e 10 dominam somente a si mesmos. ◻

Uma forma útil de apresentar um dominador é numa árvore chamada de *árvore dos dominadores*, na qual o nó inicial é a raiz e cada nó d domina somente seus descendentes na árvore. Por exemplo, a Fig. 10.14 mostra a árvore de dominadores para o grafo de fluxo da Fig. 10.13.

A existência das árvores de dominadores advém de uma propriedade dos dominadores, cada [illegible] possui um único *dominador imediato*, que é o último dominador de n sobre qualquer percurso a partir do nó inicial até n. Por razões da relação de dominação (domina, é dominado imediatamente por) possui a propriedade de que se d é o dominador imediato de n, então d *domina* n.

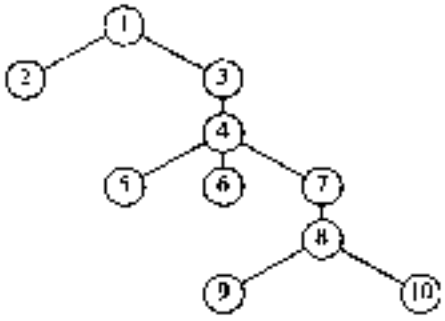


Fig. 10.14. Árvore de dominadores para o grafo de fluxo da Fig. 10.13.

Laços Naturais

Uma importante aplicação das informações sobre dominadores está na determinação dos laços adequados a melhoramentos no grau de fluxo. Existem duas propriedades essenciais desses laços:

1. Um laço precisa ter um único ponto de entrada, chamado de "cabeçalho". Esse ponto de entrada domina todos os nós do laço, pois não seria a porta de entrada, caso contrário.

2. Deve haver pelo menos uma forma de se atingir o laço, isto é, pelo menos um percurso de volta para o cabeçalho.

Uma boa forma de se encontrar todos os laços num grafo de fluxo é procurar pelos lados cujos cabeças dominam suas caudas. Desse modo, se $a \to b$ for um lado, b é o cabeçalho se b domina a. Chamaremos tais lados de *lados retrógrados*.

Exemplo 10.7. No Fig. 10.13 existem os lados $7 \to 4$ e $4 \to 3$, porque 4 domina 7 e 3 domina 4. Similarmente, $10 \to 7$ é um lado e 7 domina 10. Os outros lados com esta propriedade são $4 \to 1$, $8 \to 3$ e $9 \to 1$. Podemos ver que esses são exatamente os lados dos quais poderíamos dizer, razoavelmente, formam laços num grafo de fluxo. ∎

Dados um lado retrógrado $n \to d$, definimos um *laço natural* desse lado como sendo d mais o conjunto de nós que podem atingir n sem passar através de d. O nó d é o cabeçalho do laço.

Exemplo 10.8. O laço natural do lado $10 \to 7$ consiste nos nós 7, 8 e 10, uma vez que 8 e 10 são todos aqueles nós que podem atingir 10 sem passar através de 7. O laço natural de $4 \to 3$ é o todo o grafo de fluxo. Não se esqueça do percurso $10 \to 7 \to 8 \to 10$. ∎

Algoritmo 10.1 Construção do laço natural de um lado retrógrado.

Entrada. Um grafo de fluxo G e um lado retrógrado $n \to d$.

Saída. O conjunto *laço* consistindo em todos os nós do laço natural de $n \to d$.

Método. Começando pelo nó n, consideramos cada nó m que sabemos ser em laço, de forma a assegurar que os predecessores de m em um grafo também estejam em laço. O algoritmo é dado na Fig. 10.15. Cada nó em *laço*, exceto d, é colocado uma vez em *pilha*, de forma que seus predecessores serão examinados. Note-se que, como d é colocado no laço inicialmente, nunca examinaremos seus predecessores e, portanto, iremos encontrar aqueles nós que alcançam n sem passar através de d. ∎

```
procedimento insere(m);
se m não está em laço então início
    laço := laço ∪ {m};
    empilha m em pilha
fim;

/* início do programa principal */

pilha := vazia;
laço := {d};
insere(n);
enquanto pilha não estiver vazia faça início
    desempilha m, o primeiro elemento de pilha;
    para cada predecessor p de m faça insere(p)
fim
```

Fig. 10.15. Algoritmo para construir laço natural.

Laços mais Internos

Se usarmos os laços naturais como "laços", teremos, então, a útil propriedade de que a menos que dois laços tenham o mesmo cabeçalho, os mesmos ou serão disjuntos ou um estará inteiramente contido (um subconjunto próprio) do outro. Por conseguinte, negligenciando por enquanto os laços com o mesmo cabeçalho, temos a noção de *laço mais interno*: aquele que não contém quaisquer outros laços.

Quando dois laços possuem o mesmo cabeçalho, como na Fig. 10.16, é difícil dizer qual é o mais interno. Por exemplo, se o teste no nó de B_3 fosse

$$\ldots \to B_3 \to B_1$$

provavelmente o laço $\{B_1, B_2, B_3\}$ seria o laço mais interno. Entretanto, não poderíamos assegurar isso sem um exame detalhado do código. Talvez B_3 seja quase sempre executado após B_2, e assim o laço $\{B_1, B_3\}$ muitas vezes, antes de descer para B_2. Consequentemente, iremos supor que, quando dois laços naturais possuem o mesmo cabeçalho, nenhum dos dois estará aninhado dentro do outro, ambos serão tratados como um único laço.

Pré-Cabeçalhos

Várias transformações requerem que movamos enunciados para "antes do cabeçalho". Começamos, por conseguinte, criando um novo bloco, chamado de *pré-cabeçalho*. O pré-cabeçalho possui somente o cabeçalho como sucessor e todas as arestas que entravam anteriormente no cabeçalho de L vindas de fora de L entram agora no pré-cabeçalho. Arestas do laço de dentro de L para o cabeçalho não são modificadas. A disposição é mostrada na Fig. 10.17. Inicialmente, o pré-cabeçalho está vazio, mas as transformações com L podem colocar enunciados nele mesmo.

Grafos de Fluxo Redutíveis

Os grafos de fluxo que ocorrem na prática recaem na classe dos grafos de fluxo redutíveis, definida abaixo. O uso exclusivo de enunciados estruturados de fluxo de controle, tais como *if-then-else*, *while-do* e *break*, produz programas cujos grafos de fluxo são sempre redutíveis. Mesmo os programas escritos usando-se enunciados de desvio goto por parte de programadores sem cuidado interno pró-ativo de projeto estruturado de programas são quase sempre redutíveis.

Uma variedade de definições para "grafo de fluxo redutível" tem sido proposta. A que adotamos aqui carrega uma das noções importantes

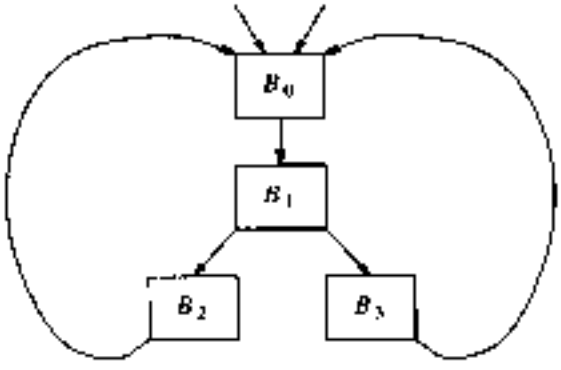


Fig. 10.16. Dois laços com o mesmo cabeçalho.

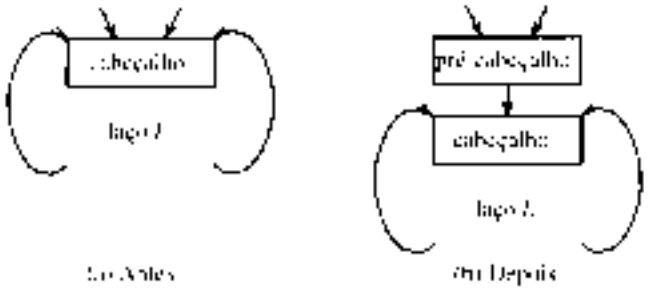


Fig. 10.17. Introdução de pré-cabeçalho.

propriedades dos grafos de fluxo redutíveis: nominalmente, não há razões para o crescimento dos laços proveniências de todos os passos, a única forma de se entrar num laço é através de seu cabeçalho. Os exercícios exploram algumas propriedades e apresentam uma breve história dos conceitos.

Um grafo de fluxo G é *redutível* se e somente se pudermos particionar os lados em dois grupos disjuntos, frequentemente chamados de lados *inflamentes* e lados *refluentes*, com as duas seguintes propriedades:

1. Os lados afluentes formam um grafo acíclico no qual cada nó pode ser atingido a partir do nó inicial de G.
2. Os lados refluentes consistem somente daqueles arcos $a \to b$ nos quais b domina a.

Exemplo 10.9. O grafo de fluxo da Fig. 10.14 é redutível. De um modo geral, se conhecermos a relação de dominação, podemos facilmente particionar o grafo de fluxo. Podemos encontrar e remover todos os lados refluentes. Os lados remanescentes têm que ser lados afluentes, se o grafo for redutível, e podemos verificar se o grafo de fluxo é redutível. Basta checar se os lados afluentes formam um grafo acíclico. No caso da Fig. 10.14, é fácil verificar que, se removermos os cinco arcos refluentes $4 \to 3$, $7 \to 4$, $8 \to 3$, $9 \to 1$ e $10 \to 7$, cujas cabeças dominam as suas caudas, o grafo remanescente é acíclico. $\square$

Exemplo 10.10. Consideremos o grafo de fluxo da Fig. 10.18, cujo enumerado é 1. Este grafo de fluxo não possui lados refluentes, já que nenhuma cabeça de um laço domina a cauda do mesmo. Por conseguinte, o grafo somente poderia ser redutível se todo o grafo fosse acíclico. Mas, como sabemos, o grafo de fluxo não é redutível. Intuitivamente, a razão pela qual este grafo de fluxo não é redutível é que o nodo 2 poderia ser inicializado de duas diferentes pontas, os nós 2 e 3. $\square$

A propriedade chave dos grafos de fluxo redutíveis para a análise de laços é que, em tais grafos de fluxo, cada conjunto de nós que consideramos intuitivamente como um laço, precisa conter um lado refluente. De fato, precisamos examinar somente os laços naturais dos lados refluentes de forma a encontrar todos os laços de um programa cujo grafo de fluxo seja redutível. Ordenadamente, o grafo de fluxo da Fig. 10.18 parece ter um "laço" envolvendo os nós 2 e 3, mas não há um lado refluente de que esse seja um laço natural. De fato, aquele "laço" possui dois cabeçalhos, 2 e 3, tornando-o mais sujeito à aplicação de muitas das técnicas de otimização de código, tais como aquelas introduzidas na Seção 10.2 para a movimentação de código e eliminação de variáveis de indução, não sejam diretamente aplicáveis.

Felizmente, as estruturas de fluxo de controle típicas encontradas em linguagens como aquelas da Fig. 10.18, aparecem tão raramente na maioria dos programas que podemos restringir nossa atenção a laços bem comportados, sem nenhuma importância secundária. Existem, de fato, linguagens como Bliss e Modula-2, que permitem apenas programas com grafos de fluxo redutíveis, e muitas outras linguagens irão produzir somente grafos de fluxo redutíveis a não ser que façam uso explícito de construções de desvio.

Fig. 10.18. Um grafo de fluxo irredutível.

Exemplo 10.11. Retornando ao nosso grafo da Fig. 10.14, notamos que o "laço mais interno" envolvendo o laço será o de blocos $\{7, 8, 10\}$, cujo laço natural do lado refluente $10 \to 7$. O conjunto $\{4, 5, 6, 7, 8, 10\}$ é o laço natural de $7 \to 4$. Note se que 8 e 10 podem atingir 7 através do lado $10 \to 7$. Nossa intuição de que $\{4, 5, 6, 7\}$ forma um laço é correta, uma vez que 4 e 7 poderiam ser ambos entradas de um laço, e todo nosso argumento de uma única entrada. Colocando de outra forma, não existe razão para assumir que o controle *gasta* muito tempo indo através dos nós $\{4, 5, 6, 7\}$ isto é, não há razão para esperar que o controle passe de 8 para 7 mais frequentemente do que o faz para 4. Pela inclusão de 8 e 10 no laço, estamos dando certo peso à tendência natural de uma região ser frequentemente atravessada no programa.

É importante reconhecer, no entanto, o perigo de se fazer suposições a respeito da frequência dos desvios. Por exemplo, se movêssemos nosso pensamento para fora do laço $\{7, 8, 10\}$ imaginando intuitivamente que este ocorre em 8 ou 10, e, de fato, o controle seguisse o laço $7 \to 4$ mais frequentemente do que $7 \to 8$, poderíamos efetivamente aumentar o número de vezes que uma dada posição seria executada. Discutiremos métodos para evitar esse problema na Seção 10.7.

O próximo laço natural é $\{3, 4, 5, 6, 7, 8, 10\}$ que é a união dos laços $4 \to 3$ e $8 \to 3$. Como antes, nossa intuição de que $\{3, 4\}$ deveria ser considerado um laço vem da exigência de que um laço tenha um cabeçalho. O último laço aqui é o para o lado refluente $9 \to 1$, é todo o grafo de fluxo. $\square$

Existem várias propriedades interessantes dos grafos redutíveis, as quais iremos introduzir quando de seu uso. Esperamos uma discussão mais profunda e a análise de intervalos na Seção 10.9.

10.3 INTRODUÇÃO À ANÁLISE GLOBAL DE FLUXO DE DADOS

Como a habilidade de realizar a otimização de um grau ou outro depende da qualidade para a geração de código, um compilador precisa coletar informações a respeito do programa como um todo e distribuí-las a cada bloco no grafo de fluxo. Por exemplo, vimos na Seção 9.7 como os conjuntos de registradores vivos na saída de cada bloco poderia melhorar a utilização dos registradores. A Seção 10.2 sugeriu a computação nos nós das expressões disponíveis com a finalidade de eliminar os cômputos redundantes. Igualmente, as Seções 9.9 e 10.2 descreveram como um computador poderia tirar vantagem das "definições incidentes", tal como quando o conhecimento do ponto local de definição de uma variável (como i em $i := i + 1$) antes de se atingir um dado bloco, poderia ser usado para a finalidade de realizar transformações como a troca posição para constantes e a otimização de código inativo. Esses fatos são simplesmente uns poucos exemplos das *informações de fluxo de dados* que um compilador pode querer capturar através de um processo conhecido por *análise de fluxo de dados*.

As informações de fluxo de dados podem ser calculadas estabelecendo e solucionando sistemas de equações que relacionam valores de variáveis em vários pontos do programa. Uma equação típica possui a forma

$$\text{saída}[s] = \text{gerada}[s] \cup (\text{entrada}[s] - \text{anulada}[s]) \quad (10.5)$$

e pode ser lida como: as informações no final da enunciado s são geradas dentro do enunciado ou entram no início e não são anuladas

medida que o controle flui através do fragmento". Tais equações são chamadas de *equações de fluxo de dados*.

Os detalhes sobre como as equações de fluxo de dados são estabelecidas e resolvidas dependem de três fatores:

1. As noções de "gerar" e "matar" dependem de uma informação que, para instâncias do problema de análise de fluxo de dados vistas neste livro, é fornecida pelas definições e referências a variáveis. Se for preciso fazer alguma análise de fluxo de dados em nível de instrução, como a análise de fluxo de controle e de fluxo de dados, precisamos processar cada instrução para extrair essa informação. Em contextos de compilação, isso é feito durante a fase de geração de código intermediário.

2. O tipo de dados desejado ao longo de percursos de controle na análise de fluxo de dados é ditado pela construção de controle do programa em questão. Em geral, quando essa construção é [A], as equações a aplicar são aquelas que exigem um único ponto final onde o controle deixa o fragmento. Em geral, as equações são estabelecidas em torno de blocos básicos em vez de instruções, porque os blocos possuem realmente pontos únicos de saída.

3. Existe a suposição de que todo o fragmento, quando chamado de procedimento, atribuições através de variáveis do tipo apropriado e pontos ambíguos são todos de interesse.

Nesta seção, consideramos o problema de determinar o conjunto de definições que alcançam um ponto em um programa e como isso pode ser computado para cada ponto de interesse. Mais tarde neste capítulo, os algoritmos para a otimização de código e a determinação de variáveis de indução também farão uso dessas informações.

Consideramos inicialmente programas construídos usando os comandos **if**-**then**-**else** e **while**. O fluxo de controle provável nesses comandos nem sempre é óbvio, mas discutimos isso mais tarde. Por ora, estabelecer e resolver as equações de fluxo de dados. As definições nesta seção são escalares ou elementos de cópia ou são de toda a [illegible]. Neste capítulo, frequentemente nos referimos a "" como um operador único. Todos os que discutimos se aplicam diretamente a outros operadores, mas não aqueles com um operando ou com mais de dois operandos.

Pontos e Percursos

Dentro de um bloco básico, à exceção do ponto entre dois comandos adjacentes, bem como do ponto antes do primeiro comando e após o último, há um ponto. Por conseguinte, o bloco B_1 na Fig. 10.19, possui quatro pontos, um antes de qualquer uma das atribuições e um após cada um dos três comandos.

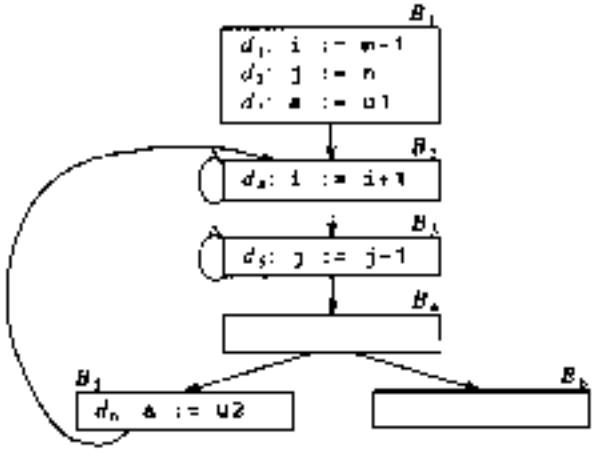


Fig. 10.19. Um grafo de fluxo.

Vamos agora numerar todos os pontos de um programa de modo que possamos nos referir a eles mais tarde. Suponha que existam n instruções no programa. Associamos o número i à instrução i, para $1 \leq i \leq n$. Associamos o número i ao ponto que precede imediatamente a instrução i, para $1 \leq i \leq n$. Também associamos o número $n+1$ ao ponto que segue imediatamente a última instrução do programa. Dizemos que a instrução i *tem* os pontos i e $i+1$, associados a ela.

Existem dois pontos especiais: o ponto 1 precede a primeira instrução do programa e o ponto $n+1$ sucede a última instrução. Se o fluxo de controle pode deixar o programa em mais de um lugar, podemos introduzir um nó fictício para o fluxo de controle, cujo predecessor são todos os pontos nos quais o fluxo de controle pode deixar o programa. Esse nó fictício tem o número $n+2$. Para simplificar, supomos que todos os programas têm um único ponto de entrada e um único ponto de saída.

Um *percurso* de um ponto p_1 até um ponto p_k é uma sequência de pontos $p_1, p_2, \ldots, p_k$ tais que, para cada i entre 1 e $k-1$, ou a instrução i tem os pontos p_i e p_{i+1} ou p_i é o fim de um bloco e p_{i+1} é o início de um sucessor desse bloco.

Exemplo 10.12. Na Fig. 10.19, existe um percurso do ponto 2 do bloco B_2 para o início do bloco B_4. Consiste na instrução 2, do ponto final de B_2, e, em seguida, através de todos os pontos em B_3, B_4 e B_5 até o início de B_6. ☐

Definições Incidentes

Uma *definição* de uma variável x é uma instrução que atribui, ou pode atribuir, um valor a x. As duas formas mais comuns de definições são as instruções que incondicionalmente atribuem um valor a x, como a instrução de entrada e saída, e as instruções que atribuem um valor a x se um determinado endereço for o endereço de x. Essas instruções definem um valor para x e são referenciadas como definições *inequívocas* de x. Existem certas construções de endereço, como os que podem definir um valor para x, são chamadas de definições *ambíguas*. As formas mais comuns de instruções ambíguas de [illegible]:

1. Uma atribuição através de um apontador que pode referenciar x. Por exemplo, a atribuição $*q := y$ é uma definição de x se for possível que q aponte para x. Os métodos para se determinar para o que um apontador pode apontar são discutidos na Seção 10.6. Uma atribuição de qualquer componente de uma variável que é uma matriz, estrutura ou registro também é considerada uma definição de cada variável.

2. Uma atribuição através de um apontador que pode referenciar x. Por exemplo, a atribuição $*p := y$ é uma definição de x se for possível que p aponte para x. Os métodos para se determinar para o que um apontador pode apontar são discutidos na Seção 10.6. Uma atribuição de qualquer componente de uma variável que é uma matriz, estrutura ou registro também é considerada uma atribuição através de um apontador e uma definição de cada variável.

Dizemos que uma definição d *alcança* um ponto p se existe um percurso do ponto imediatamente após d até p tal que d não seja "morta" ao longo do percurso, isto é, não exista uma definição d' de x em uma variável x incidente em p. Uma definição pode alcançar um ponto mesmo se ela for morta ao longo de algum percurso. Mas se houver uma definição d' de x incidente em p, d' pode ser eliminada antes de p ou depois de p, mas não em ambos os lugares. Também dizemos que uma definição d alcança o início de um bloco B se d alcança um ponto imediatamente antes de uma instrução de B. Por exemplo, a definição d_1 na Fig. 10.19 alcança o início do bloco B_2, assim como também alcançam as definições d_2 e d_3. Entretanto, nenhuma definição em B_2, B_3 ou B_4 alcança o início de B_2, pois todas são mortas se B_1 for executado antes de B_2 e qualquer uma delas for executada.

Vamos agora considerar o grafo de fluxo global e considerar todos os pontos em todos os blocos. Um percurso $p_1, p_2, \ldots, p_k$ é uma sequência de pontos $p_1, p_2, \ldots, p_k$ tais que, para cada i entre 1 e $k-1$, uma das duas situações abaixo é válida:

1. p_i é o ponto que precede imediatamente a uma instrução e p_{i+1} é o ponto que se segue imediatamente àquela instrução no mesmo bloco básico.
2. p_i é o final de algum bloco e p_{i+1} é o início de um bloco sucessor.

Exemplo 10.12. Na Fig. 10.19, existe um percurso do ponto 2 do bloco B_2 para o início do bloco B_4. Consiste na instrução 2, do ponto final de B_2, e, em seguida, através de todos os pontos em B_3, B_4 e B_5 até o início de B_6. ☐

Definições Incidentes

Uma *definição* de uma variável x é uma instrução que atribui, ou pode atribuir, um valor a x. As duas formas mais comuns de definições são as instruções que incondicionalmente atribuem um valor a x, como a instrução de entrada e saída, e as instruções que atribuem um valor a x se um determinado endereço for o endereço de x. Essas instruções definem um valor para x e são referenciadas como definições *inequívocas* de x. Existem certas construções de endereço, como os que podem definir um valor para x, são chamadas de definições *ambíguas*. As formas mais comuns de instruções ambíguas de [illegible]:

1. Uma atribuição através de um apontador que pode referenciar x. Por exemplo, a atribuição $*q := y$ é uma definição de x se for possível que q aponte para x. Os métodos para se determinar para o que um apontador pode apontar são discutidos na Seção 10.6. Uma atribuição de qualquer componente de uma variável que é uma matriz, estrutura ou registro também é considerada uma atribuição através de um apontador e uma definição de cada variável.

2. Uma atribuição através de um apontador que pode referenciar x. Por exemplo, a atribuição $*p := y$ é uma definição de x se for possível que p aponte para x. Os métodos para se determinar para o que um apontador pode apontar são discutidos na Seção 10.6. Uma atribuição de qualquer componente de uma variável que é uma matriz, estrutura ou registro também é considerada uma atribuição através de um apontador e uma definição de cada variável.

Dizemos que uma definição d *alcança* um ponto p se existe um percurso do ponto imediatamente após d até p tal que d não seja "morta" ao longo do percurso, isto é, não exista uma definição d' de x em uma variável x incidente em p. Uma definição pode alcançar um ponto mesmo se ela for morta ao longo de algum percurso. Mas se houver uma definição d' de x incidente em p, d' pode ser eliminada antes de p ou depois de p, mas não em ambos os lugares. Também dizemos que uma definição d alcança o início de um bloco B se d alcança um ponto imediatamente antes de uma instrução de B. Por exemplo, a definição d_1 na Fig. 10.19 alcança o início do bloco B_2, assim como também alcançam as definições d_2 e d_3. Entretanto, nenhuma definição em B_2, B_3 ou B_4 alcança o início de B_2, pois todas são mortas se B_1 for executado antes de B_2 e qualquer uma delas for executada.

Vamos agora considerar o grafo de fluxo global e considerar todos os pontos em todos os blocos. Um percurso $p_1, p_2, \ldots, p_k$ é uma sequência de pontos $p_1, p_2, \ldots, p_k$ tais que, para cada i entre 1 e $k-1$, uma das duas situações abaixo é válida:

1. p_i é o ponto que precede imediatamente a uma instrução e p_{i+1} é o ponto que se segue imediatamente àquela instrução no mesmo bloco básico.
2. p_i é o final de algum bloco e p_{i+1} é o início de um bloco sucessor.

Definições desse tipo, permitirmos algumas imprecisões na computação das definições mesmo-nós. Se, em vez, todas as computações estão na direção "segura" ou "conservativa". Como exemplo, notemos que nosso suposição de que todos os caminhos de um grafo de fluxo possam ser percorridos. Isso pode não ser verdadeiro na prática. Por exemplo, para certos valores de x e de y, pode ser impossível atingir efetivamente o enunciado $z = 1$ no seguinte fragmento de programa.

```
if x > y then
    z = 1
```

Em geral, decidir se cada percurso no grafo de fluxo pode ser seguido em uma execução do programa é um problema insolúvel. Daqui

Uma outra ocorrência no projeto de transformações para melhorar o código é que, se somos forçados a rejeitar apenas decisões conservativas ao determinarmos com qualquer dúvida, apesar das estratégias conservativas nos fazerem perder algumas oportunidades de fato poderíamos realizar seguramente. Uma decisão conservativa nunca levará a uma mudança do que o programa computa. Nas aplicações de definições incidentes é normalmente conservativo assumir que uma definição atinge um ponto, ainda que de fato não possa fazê-lo. Por conseguinte, permitimos os percursos que ainda venham a ser percorridos em qualquer execução do programa, e permitimos que passem definições através de definições ambíguas da mesma variável.

Análise de Fluxo e Dados de Programas Estruturados

Os grafos de fluxo para construções de fluxo de controle tais como enunciados *do-while* possuem uma propriedade útil: existe um único ponto de início através do qual o controle entra e um único ponto de saída através do qual o controle deixa quando a execução do comando está terminada. Exploramos esta propriedade quando falamos das definições incidentes ao início e fim de enunciados, que tem a seguinte sintaxe:

$S \rightarrow$ **id** $=$ E | S ; S | **if** E **then** S **else** S | **do** S **while** E.
$E \rightarrow$ **id** + **id** | **id**

As expressões nesta linguagem são similares àquelas no código intermediário, mas os grafos de fluxo para enunciados têm formas restritas que são sugeridas pelos diagramas na Fig. 10.20. Um propósito principal desta seção é o de estabelecer as equações de fluxo de dados sugeridas na Fig. 10.21.

Definimos uma *região* de grafo de fluxo chamada de uma região pelo cabeçalho, o único nó no conjunto de nós N que pode ser atingido a partir de arcos do grafo de fluxo que começam fora de N. Dizemos que o nó cabeçalho *domina* todos os nós em N, pois todos os caminhos de início do grafo até qualquer nó em N devem passar pelo cabeçalho. A região do grafo de fluxo correspondente a um enunciado é a região que obedece à restrição adicional de que nenhum nó, exceto o cabeçalho, tenha arcos de entrada ou de saída. Intuitivamente, a região corresponde a um bloco executado quando entra a região.

Como uma conveniência técnica, assumimos que existam blocos fictícios sem comandos, indicados por círculos abertos na Fig. 10.20, através dos quais o controle flui imediatamente antes de entrar e deixar a região. Dizemos que os pontos de início dos blocos fictícios a entrada e saída de um enunciado da região são *pontos em nível final* respectivamente do enunciado.

As equações na Fig. 10.21 são definições indutivas, ou dirigidas pela sintaxe, dos conjuntos *entrada*[S], *saída*[S] e *gerados*[S] para todos os enunciados S. Os conjuntos *entrada*[S] e *saída*[S] são atributos sintetizados; são computados de baixo para cima a partir dos nós em árvores sintáticas para os enunciados. Nosso desejo é que a definição d esteja em *gerados*[S], se d atingir o final de S independentemente de ainda

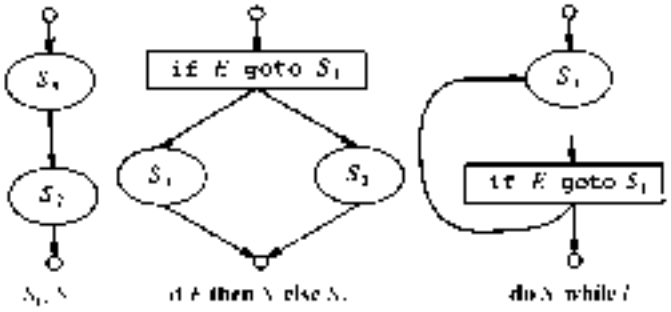


Fig. 10.20. Algumas construções e símbolos estruturados

estar ou não em *entrada*[S]. Colocando de outra forma, d precisa aparecer em S e atingir o final de S através de um percurso que não vai para fora de S. Este requisito garante que *gerados*[S] é o conjunto de definições "geradas por S".

Similarmente, desejamos que *mortos*[S] seja o conjunto de definições que nunca atingem o final de S, mesmo que atinjam o início. Consequentemente, *mortos* sempre diverge das definições que são mortas por S. Para a definição d estar em *mortos*[S], cada percurso para o enunciado final de S precisa ter uma definição intermediária da mesma variável definida por d, se d aparecer em S; então, seguindo cada sugestão de que ao longo de qualquer percurso se tenha encontrado uma definição da mesma variável.

As regras para *gerados* e *mortos* serão traduções simples. Considere uma definição de x em um enunciado S e suponha que x não é definido em nenhum lugar em S. Então, a definição de x em S é gerada por S se e somente se S não mata x. Certamente, aquela atribuição é uma definição que atinge o final do enunciado independentemente de atingir ou não o início. Por conseguinte,

$$gerados[S] = \{d\}$$

Por outro lado, d mata todas as outras definições de x em S. Por isso escrevemos

$$mortos[S] = D - \{d\}$$

onde D é o conjunto de todas as definições no programa para a variável x.

A regra para enunciados compostos, ilustrada na Fig. 10.21(b), é um tanto mais sutil. Sob que circunstâncias é uma definição d gerada por $S = S_1; S_2$? Antes de mais nada, se for gerada por S_2, então certamente é gerada por S. Se d é gerada por S_1, mas atinge o final de S_1 não for morta por S_2. Por conseguinte, escrevemos

$$gerados[S] = gerados[S_2] \cup (gerados[S_1] - mortos[S_2])$$

Um raciocínio similar se aplica à morte de uma definição e então temos

$$mortos[S] = mortos[S_1] \cup (mortos[S_2] - gerados[S_1])$$

Para o enunciado if ilustrado na Fig. 10.21(c), notamos que se for executado um desvio condicional E gera uma definição, a mesma que vai ao fim do enunciado S. Então,

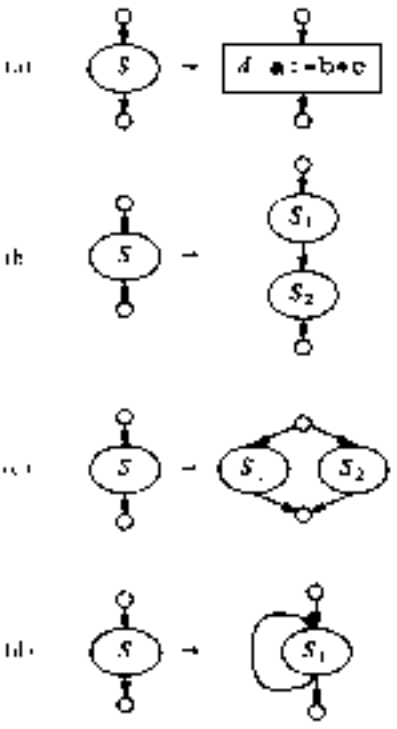
$$gerados[S] = gerados[S_1] \cup gerados[S_2]$$

Se entrarmos a fim de "matar" uma definição d, a variável definida por d precisa ser morta ao longo de qualquer percurso partindo do início

---

Embora em um certo sentido a região que forma o enunciado *do-while* tenha dois nós cabeçalho, o início do corpo e o nó de teste, o início do corpo é o único nó que pode ser atingido a partir de fora do enunciado.

Note que também estamos assumindo que todos os caminhos em um grafo de fluxo podem ser percorridos. Isso não é verdade, mas necessitamos de algoritmos



$$gerados[S] = \{A\}$$
$$mortos[S] = \{D \mid \ldots = \ldots A \ldots\}$$

$$saídos[S] = gerados[S] \cup (entrados[S] - mortos[S])$$

$$gerados[S] = gerados[S_2] \cup (gerados[S_1] - mortos[S_2])$$
$$mortos[S] = mortos[S_2] \cup (mortos[S_1] - gerados[S_2])$$

$$entrado[S_1] = entrado[S]$$
$$entrado[S_2] = saído[S_1]$$
$$saído[S] = saído[S_2]$$

$$entrado[S] = gerados[S_1] \cup gerados[S_2]$$
$$mortos[S] = mortos[S_1] \cap mortos[S_2]$$

$$entrado[S_1] = entrado[S]$$
$$entrado[S_2] = entrado[S]$$
$$saído[S] = saído[S_1] \cup saído[S_2]$$

$$gerados[S] = gerados[S_1]$$
$$mortos[S] = mortos[S_1]$$

$$entrado[S_1] = entrado[S] \cup gerados[S_1]$$
$$saído[S] = saído[S_1]$$

Fig. 10.21 Equações de fluxo de dados para as definições recentes.

ponto final de S. Em particular, precisa ser igual ao longo de um ciclo que envolva S, dessa forma:

$$mortos[S] = mortos[S_2] \cap mortos[S_1]$$

Para uma definição d gerada dentro de S, a regra para $gerados$ é clara. Colocando de forma simples, a definição d é gerada em $gerados$ se e somente se a definição d é gerada dentro de S e não é morta no final de S. Por outro lado, se d é gerada dentro de S, pode ser morta se for gerada dentro de S_2. Se d é morta, por S_2, ao longo do caminho que leva a variável de d a ser redefinida dentro de S_2, a cada vez. Por outro lado, se d é morta por S_1, então é morta seguramente por S. Concluímos que

$$gerados[S] = gerados[S_2] \cup (gerados[S_1] - mortos[S_2])$$
$$mortos[S] = mortos[S_2] \cap mortos[S_1]$$

Estimativa Conservativa das Informações de Fluxo de Dados

Existe um erro sutil de cálculo por *gerados* e *mortos* dados na Fig. 10.21. Levantei a suposição de que a expressão condicional E nas construções *if-then-else* e *while* é "não interpretada", o que quer dizer que existe um caminho para cada uma das duas escolhas dessas construções para uma outra decisão. Colocando de outra forma, assumimos que qualquer percurso em grafo sintático no diagrama de fluxo é também um percurso de execução; isto é, um percurso que S executado quando o programa é rodado com pelo menos uma possível entrada.

Este não é sempre o caso e, de fato, em geral não podemos decidir se uma ramificação será seguida. Suponhamos, por exemplo, que a expressão E nunca produza *true* quando for executada. Então, o percurso ao longo de S_1 na Fig. 10.21(c) não poderia jamais ser seguido. Isto contradiz nossa suposição. Porque uma definição gerada por S_1 não é realmente gerada por S porque não há como se ir do início de S para o término de S. Segunda, nenhuma definição em $mortos[S]$ poderia ainda estar no final de S, pois sempre, tal definição deveria logicamente estar em $mortos[S]$, mesmo que não esteja em $mortos[S]$.

Quando comparamos os conjuntos $gerados$ computados com o conjunto $gerados$ "verdadeiro", descobrimos que o $gerados$ "verdadeiro" é sempre um subconjunto de $gerados$ computado. Por outro lado, o conjunto $mortos$ verdadeiro é sempre um superconjunto do conjunto $mortos$ computado. Essas relações de inclusão são válidas mesmo depois de considerarmos as outras regras da Fig. 10.21. Por exemplo, se a expressão E num enunciado *do S-while E* também for falsa, podemos jamais sair do laço. Por conseguinte, os $gerados$ verdadeiro e computado de um laço é nulo pelo laço. O caso dos enunciados inversos na Fig. 10.21(c) também precisa ser levado em conta, mas a impossibilidade de se sair de S_1 ou S_2 por causa de um laço infinito é de novo como antes.

É natural perguntar se essas diferenças entre os conjuntos gerados e mortos verdadeiros e computados apresentam uma séria substituição à análise do fluxo de dados. A resposta reside no uso pretendido para esses dados. No caso particular de definições recentes, usamos principalmente as definições para inferir que o valor de uma variável x a um ponto está limitado a algumas possibilidades dentre pequenos números. Por exemplo, se concluirmos que as únicas definições de x que alcançam aquele ponto são da forma $x := 1$, podemos inferir que x possui o valor 1 naquele ponto. Por conseguinte, poderíamos decidir substituir as referências a x por referências a 1.

Como consequência, se perdermos um conjunto de definições que alcançam um determinado ponto não parece ser muito sério, inicialmente nos impede de realizar uma otimização que poderíamos de outra forma fazer. Por outro lado, suponhamos o conjunto de definições é um erro fatal pois nos leva a realizar uma mudança no programa a qual nenhuma troca o programa comprova. Por exemplo, podemos pensar que todas as definições recentes de x conferem ao mesmo valor 1 e assim equivocadamente substituir x por 1 mas pode haver uma outra definição recente, não detectada, que dá x o valor 2. Para o caso das definições recentes, denominamos um conjunto de definições seguro ou *conservativo* se a estimativa for um superconjunto (não necessariamente próprio) do conjunto verdadeiro das definições recentes. Chamamos a definição de *pequena* se não for necessariamente um superconjunto do conjunto verdadeiro.

Para cada problema de fluxo de dados precisamos examinar o efeito das estimativas inacuradas sobre os tipos de modificações de programa que as mesmas podem causar. Aceitamos geralmente as discrepâncias que sejam seguras, no sentido de que as mesmas possam proibir otimizações que poderiam ser legalmente feitas, mas não aceitamos as que sejam não seguras, no sentido em que possam causar "otimizações" que não preservem o comportamento do programa observado externamente. Em cada problema de fluxo de dados, um subconjunto ou um superconjunto da resposta verdadeira, mas não ambos, é usualmente seguro.

Retornando agora às implicações de segurança na estimativa de *gerado* e *morto* para as definições incidentes, notemos que nossas discrepâncias, superconjuntos para *gerado* e subconjuntos para *morto*, estão ambas na direção segura. Intuitivamente, aumentar *gerado* adiciona ao conjunto definições que podem atingir um ponto e não pode impedir que uma definição atinja um local que a mesma efetivamente atingiu. Igualmente, decrementar *morto* pode somente aumentar o conjunto de definições incidentes a um dado ponto.

Cômputo de *Entrada* e *Saída*

Muitos problemas de fluxo de dados podem ser resolvidos por traduções sintetizadas similares àquelas usadas para se computar *gerado* e *morto*. Por exemplo, podemos querer determinar, para cada enunciado S, o conjunto das variáveis que sejam definidas dentro de S. Essa informação pode ser computada por equações análogas àquelas para o cômputo de *gerado*, sem mesmo serem requeridos conjuntos semelhantes a *morto*. Essa informação pode ser usada para determinar os cômputos laço-invariantes.

No entanto, existem outros tipos de informações de fluxo de dados, tais como os do problema das definições incidentes, que usamos como exemplo, onde também necessitamos computar certos atributos herdados. Acontece que *entrada* é um atributo herdado e *saída* é um atributo sintetizado dependente de *entrada*. Pretendemos que *entrada*[S] seja o conjunto de definições que atingem o início de S, levando em conta o fluxo de controle através de todo o programa, incluindo os enunciados fora de S ou dentro dos quais S esteja embutido. O conjunto de *saída*[S] é definido similarmente para o final de S. É importante notar uma distinção entre *saída*[S] e *gerado*[S]. O último é o conjunto de definições que atingem o final de S sem seguir por percursos fora de S.

Como um exemplo simples da diferença, consideremos os enunciados em cascata na Fig. 10.21(b). Um enunciado d pode ser gerado em S_1 e, consequentemente, atingir o início de S_2. Se d não estiver em *morto*[S_2], d atingirá o final de S_2 e, consequentemente, estará em *saída*[S_2]. No entanto, d não está em *gerado*[S_2].

Após computar *gerado*[S] e *morto*[S], de baixo para cima, para todos os enunciados S, devemos computar *entrada* e *saída*, começando pelo enunciado que representa o programa completo, compreendendo que *entrada*[S_0] = ∅ se S_0 for o programa completo. Isto é, nenhuma definição atinge o início do programa. Para cada um dos quatro tipos de enunciados da Fig. 10.21, podemos assumir que *entrada*[S] seja conhecido. Precisamos usá-lo para computar *entrada* para cada um dos subenunciados de S (o que é trivial nos casos (b)-(d) e irrelevante no caso (a)). Em seguida, computamos recursivamente (de cima para baixo) *saída* para cada um dos subenunciados S_1 ou S_2 e usamos esses conjuntos para computar *saída*[S].

O caso mais simples é o da Fig. 10.21(a), onde o enunciado é uma atribuição. Assumindo que conheçamos *entrada*[S], computamos *saída* através da equação (10.5), isto é,

$$saída[S] = gerado[S] \cup (entrada[S] - morto[S])$$

Dito em palavras, uma definição atinge o final de S se for gerada por S (isto é, for a definição d que é o enunciado), ou se atinge o início do enunciado e não for morta pelo mesmo.

Suponhamos ter computado *entrada*[S] e que S seja os dois enunciados em cascata S_1; S_2, como no segundo caso da Fig. 10.21. Começamos observando *entrada*[S_1] = *entrada*[S]. Em seguida, computamos recursivamente *saída*[S_1], que fornece *entrada*[S_2], uma vez que uma definição atinge o início de S_2 se e somente se atingir o final de S_1. Podemos agora computar recursivamente *saída*[S_2], e esse conjunto é igual a *saída*[S].

Em seguida, consideremos o enunciado *if* da Fig. 10.21(c). Como fizemos, assumimos conservadoramente que o controle poderia seguir uma ou outra ramificação; uma definição atinge o início de S_1 ou S_2 exatamente quando atinge o início de S. Isto é,

$$entrada[S_1] = entrada[S_2] = entrada[S]$$

Também se segue a partir do diagrama na Fig. 10.21(c) que uma definição atinge o final de S se e somente se a mesma atingir o final de um dos subenunciados; isto é,

$$saída[S] = saída[S_1] \cup saída[S_2]$$

Podemos, por conseguinte, usar essas equações para computar *entrada*[S_1] e *entrada*[S_2] a partir de *entrada*[S], para computar recursivamente *saída*[S_1] e *saída*[S_2] e usá-los para computar *saída*[S].

Lidando com Laços

O último caso, a Fig. 10.21(d), apresenta problemas especiais. Vamos assumir novamente que nos sejam dados *gerado*[S_1] e *morto*[S_1], tendo os mesmos sido computados de baixo para cima, e vamos assumir que nos seja dado *entrada*[S], na medida em que estejamos no processo de travessia em profundidade da árvore sintática. Diferentemente dos casos (b) e (c), não podemos simplesmente usar *entrada*[S] como *entrada*[S_1], porque as definições dentro de S_1 que atingem o final de S_1 são capazes de seguir o arco de volta para o início de S_1 e, por conseguinte, também estão em *entrada*[S_1]. Em vez disso, temos

$$entrada[S_1] = entrada[S] \cup saída[S_1] \qquad (10.6)$$

Temos também a equação óbvia para *saída*[S]:

$$saída[S] = saída[S_1]$$

a qual poderemos usar uma vez que tenhamos computado *saída*[S_1]. No entanto, parece que não podemos computar *entrada*[S_1] por (10.6) até que tenhamos computado *saída*[S_1], e o nosso plano geral foi computar *saída* para um enunciado computando primeiro *entrada* para o mesmo.

Felizmente, existe uma forma direta de se expressar *saída* em termos de *entrada*; é dada por (10.5) ou, neste caso particular,

$$saída[S_1] = gerado[S_1] \cup (entrada[S_1] - morto[S_1]) \qquad (10.7)$$

É importante compreender o que está acontecendo aqui. Não sabemos realmente que (10.7) seja verdadeira a respeito de um enunciado arbitrário S_1; apenas suspeitamos que deveria ser verdadeira porque "faz sentido" que uma definição deva atingir o final de um enunciado se e somente se tiver sido gerada dentro do mesmo ou se atingir o seu início e não for morta dentro do mesmo. No entanto, a única maneira de se saber como computar *saída* para um enunciado é através das equações fornecidas na Fig. 10.21(a)-(c). Vamos assumir (10.7) e derivar as equações para *entrada* e *saída* na Fig. 10.21(d). Podemos em seguida usar as equações da Fig. 10.21(a)-(d) para provar que (10.7) é válida para um S_1 arbitrário. Poderíamos, em seguida, colocar essas provas juntas para realizar uma prova por indução no comprimento do enunciado S_1 e de todos os seus subenunciados. Não faremos tal, deixando as provas como exercícios, mas o raciocínio que seguimos aqui deve se provar instrutivo.

Mesmo assumindo [10.6] e [10.7], a solução existe e é única (ver exercício 10.8). Essas duas equações definem uma função para cada $n \in N_t$ (exceto N_1) simultaneamente. Vamos escrever as equações como

$$\begin{aligned} I &= \lambda(O) \\ O &= f_n(I) = \mathrm{gen}_n \cup (I - \mathrm{kill}_n) \end{aligned} \qquad [10.8]$$

onde I, O, λ, $f_n \in H$ correspondem a $entrada[N_t]$, $saída[N_t]$, $entrada[N]$, $gen_n[N]$, $saída[N]$ e $kill_n[N]$, respectivamente. Os dois primeiros são valores vetoriais e os demais são constantes.

Para resolver [10.8], vamos assumir que $\lambda = \emptyset$. Podemos então substituir a primeira equação em [10.8], para compor uma estimativa de custo:

$$I = f$$

Em seguida, podemos usar a segunda equação para obter uma definição recursiva de O:

$$O^0 = \lambda \qquad O^{k+1} = f_n(O^k) = \mathrm{gen}_n \cup (O^k - \mathrm{kill}_n)$$

A aplicação da primeira equação a cada nó resulta em O como sendo

$$I^k = \lambda \cup O^k = \lambda \cup f_n(O^k) = \lambda \cup \mathrm{gen}_n \cup (O^k - \mathrm{kill}_n)$$

Se substituirmos a segunda equação, a primeira estimativa de O será

$$O^1 = \mathrm{gen}_n \cup (O^0 - \mathrm{kill}_n) = \mathrm{gen}_n \cup (\emptyset - \mathrm{kill}_n) = \mathrm{gen}_n$$

Note-se que $O^0 = O$. Por conseguinte, se computarmos a primeira estimativa de O, a mesma será igual a I, o que nos dará uma outra estimativa de O igual O e assim por diante. Por conseguinte, os valores iniciais para I e O são aqueles fornecidos para I^0 e O^0. Com duas variáveis, então, as equações da Fig. 10.21 (i) que são

$$\begin{aligned} entrada[N_t] &= entrada[N_t] \cup saída[N_t] \\ saída[N_t] &= saída[N_t] \end{aligned}$$

A vantagem dessas equações é que elas podem ser computadas em uma única passagem seguindo o exame do grafo G_t (Fig. 10.21 (ii)).

Um detalhe que ainda é preciso tratar diz respeito à inicialização com $O = \emptyset$. Relembremos que em nossas discussões das estruturas conservativas sugerimos que conjuntos como $entrada[N_t]$, em lugar dos quais O está, deveriam ser representados através de vetores de bits. De fato, se fizermos a suposição usual com $O = \{d_1\}$, pode-se ter uma definição que não apareça nem em I (ou H, dependendo de para onde os valores iniciais forem) e O quando de I quando de O.

Agora, precisamos discutir os significados pretendidos para $entrada$ e $saída$. Se uma definição d realmente pertencesse a $entrada[N_t]$, deveria existir um percurso até o ponto de ponto que permitisse que d estivesse definida no único de N_t que realmente corresponde a alguma dentre esse ponto. Se d estivesse fora de N então d teria que estar em $entrada[N]$ enquanto que se d estivesse dentro de N por consequinte não teria que estar em S_t teria que estar em $saída[N]$. Se nenhum caso estiver em I e, consequentemente, calculado em I por [10.8]. No segundo caso, d seria dito em G e, de novo, transmitido para G através de O em [10.8]. A conclusão é que começando com uma estimativa pequena, construída de acordo com [10.6] e [10.7], uma iteração para I e O cuida fora a seguinte de estimar $entrada[N_t]$

Representação dos Conjuntos

Definições de conjuntos, tais como $gerados[N]$, $entrada[N]$, podem ser representadas eficientemente usando-se vetores de bits. Atribuímos um número para cada definição de interesse no grafo de fluxo. O vetor de bits que representa um conjunto de definições terá um bit ligado se e somente se uma definição correspondente estiver no conjunto.

O método de implementação de ações que podem ser tomadas quando um comando é encontrado é muito parecido com o que usamos para os comandos. No entanto, nem todas as definições podem ser de interesse global ao bloco de dados. Por exemplo, definições de temporários que serão usadas somente dentro de uma única instrução não serão mantidas através dos temporários gerados para a validação de expressões nos processos semânticos atribuídos. Por conseguinte, as mínimas definições de interesse serão tipicamente registradas numa tabela separada.

A representação sob a forma de vetores de bits por nós também permite que operações sobre conjuntos sejam implementadas eficientemente. Vamos relembrar que dois conjuntos podem ser representados pelos mesmos operadores lógicos respectivamente, com exceção feita ao complemento. As operações lógicas correspondentes para $A \cup B$ e $A \cap B$ são, respectivamente, A OR B e A AND B. A diferença $A - B$ dos conjuntos A e B pode ser implementada fazendo-se o complemento de B e, em seguida, tomando-se o AND lógico com A.

Exemplo 10.13. A Fig. 10.22 mostra um programa com sete definições numeradas para d_1 a d_7 nos comentários à esquerda das definições. Vetores de bits representando os conjuntos $gerados$ e $mortos$ para os enunciados da Fig. 10.22 são mostrados à esquerda dos enunciados na Fig. 10.23. Os conjuntos entrada e saída são computados através da aplicação das equações da Fig. 10.21 aos enunciados representados pelos dados da Fig. 10.23.

Consideremos o enunciado do caixa 3 da Fig. 10.23. O conjunto $entrada$ é representado por 0000001 e o conjunto $mortos$ por 1001000. Se d_1 é uma das definições que chegam ao enunciado, então d_1 e d_4 também chegam. Isso é representado pelo vetor de bits 1001001. Se d_2 é uma das definições que chegam ao enunciado, então d_2 e d_5 também chegam. Isso é representado pelo vetor de bits 0100101. Se d_3 é uma das definições que chegam ao enunciado, então d_3 e d_6 também chegam. Isso é representado pelo vetor de bits 0010011. Se d_4 é uma das definições que chegam ao enunciado, então d_4 também chega. Isso é representado pelo vetor de bits 0001000. Se d_5 é uma das definições que chegam ao enunciado, então d_5 também chega. Isso é representado pelo vetor de bits 0000100. Se d_6 é uma das definições que chegam ao enunciado, então d_6 também chega. Isso é representado pelo vetor de bits 0000010. Se d_7 é uma das definições que chegam ao enunciado, então d_7 também chega. Isso é representado pelo vetor de bits 0000001. Os conjuntos $entrada$ e $saída$ são computados iterativamente, começando com $entrada = 0000000$ para todos os enunciados.

Os segundo e terceiro filhos do nó if representam as partes da direita e da esquerda, respectivamente, do comando condicional. Note-se que o conjunto $gerados$ 0000011 é a união dos conjuntos 0000010 e 0000001 nos segundo e terceiro filhos. O conjunto $mortos$ é vazio porque os conjuntos de definições mortas pelas partes direita e esquerda são disjuntos.

As equações de fluxo de dados para enunciados em caixas são calculadas com base em [10.6]. O conjunto $entrada$ desse nó é dado por

$$\mathrm{entrada}_3 = \mathrm{saída}_2 \cup \mathrm{saída}_1 = 0001011 \cup 0010011 = 0011011$$

Em palavras, isso é o mesmo que o conjunto $saída$ do enunciado 2 mais o conjunto $saída$ do enunciado 1, o gerado pelo enunciado 3 e o conjunto $entrada$ desse enunciado menos d_3 e d_6, que estão no conjunto $mortos$ do enunciado 3.

Podemos agora computar $entrada$ e $saída$ para cada enunciado, como segue. Assumimos que os conjuntos $entrada$ estão na ordem da numeração dos enunciados. Consequentemente, sabemos que os filhos esquerdo e direito de um $seqüência$ são computados antes do $seqüência$. Este também é o caso dos conjuntos $entrada$ antes do if. A partir das equações de fluxo de dados usadas

```
/* d1 */    i := m-1;
/* d2 */    j := n;
/* d3 */    a := u1;
            do
/* d4 */        i := i+1;
/* d5 */        j := j-1;
            if e1 then
/* d6 */            a := u2
            else
/* d7 */            i := u3
            while e2
```

Fig. 10.22. Um programa para demonstrar definições mortas

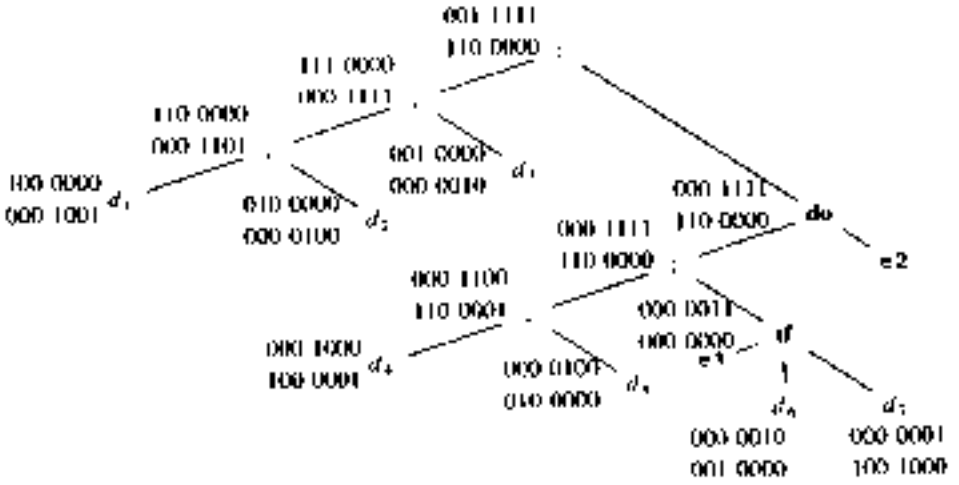


Fig. 10.21 Os conjuntos gerados e mantidos nos nós de uma árvore sintática.

à produção do nó. Fig. 10.21, o conjunto vazio após o enunciado dentro do laço **do** é obtido removendo-se a única definição do enunciado 111 0000 do conjunto gerado para 000 1111 no próximo enunciado. A união produz 111 1111, e dessa forma todas as definições atingem o início do corpo do laço. No entanto, exatamente no ponto antes da definição d_3 o conjunto mantido é 111 1110 uma vez que a definição d_3 mata d_3 (cf. 6). A análise dos conjuntos de entrada e saída deixada como um exercício.

Definições Incidentes Locais

O espaço para as informações de fluxo de dados pode ser negociado em troca de tempo, salvando-se informações somente em determinados pontos e, conforme necessário, recomputando-se as demais em determinados pontos intermediários. Os blocos básicos são uma escolha muito conveniente de unidade durante a análise de fluxo de dados global, com a atenção restrita somente àqueles pontos que sejam início de blocos. Como usualmente existem muito mais nós do que blocos, restringir nossas atenções aos blocos se constitui numa economia significativa. Quando necessário, as definições incidentes para todos os pontos de um bloco podem ser computadas a partir das definições incidentes para o início do bloco.

Em mais detalhes, consideremos uma sequência de atribuições $S_1, S_2, \ldots, S_n$ num bloco básico B. Referimo-nos ao início de B como o ponto p_1 e ao ponto entre os enunciados S_{i-1} e S_i como o ponto p_i. O final do bloco é o ponto p_{n+1}. As definições que atingem o ponto p_i podem ser obtidas a partir de *entrada*[B], considerando-se os enunciados $S_1, S_2, \ldots, S_{i-1}$ e começando-se com S_1, e aplicando-se as equações de fluxo de dados da Fig. 10.21 para enunciados em cascata. Inicialmente, seja D = *entrada*[B]. Quando S_j é considerado, removemos de D as definições mortas por S_j e adicionamos as definições geradas por S_j. Ao final, D conterá todas as definições incidentes em p_i.

Cadeias Uso-Definição

É frequentemente conveniente armazenar as informações sobre as definições incidentes sob a forma de "cadeias uso-definição" ou cadeias-ud, que são listas de todas as definições que atingem aquele uso para cada uso de uma variável. Se o uso de uma variável a num bloco B não é precedido por definições inambíguas de a, a cadeia-ud para aquele uso de a é o conjunto de definições em *entrada*[B] que sejam definições de a. Se existem definições inambíguas de a dentro de B precedendo o uso de a, então somente a última de tais definições deve estar na cadeia-ud para o uso de a em *entrada*[B] não é colocado na cadeia-ud. Adicionalmente, se existem definições ambíguas de a, então todas aquelas para as quais nenhuma definição inambígua de a existe entre o início do nó e o uso de a estarão na cadeia-ud para esse uso de a.

Ordem de Avaliação

As técnicas para conservar espaço durante a avaliação de atributos discutidas no Capítulo 5 também se aplicam ao computo das informações de fluxo de dados, usando-se especificações como aquela da Fig. 10.21. Especificamente, a única restrição sobre a ordem de avaliação dos conjuntos *gerados*, *mortos*, *entrada* e *saída* para os enunciados é aquela imposta pelas dependências entre os atributos. Tendo-se feito uma ordem de avaliação, estamos livres para liberar o espaço para o conjunto após todos os usos do mesmo terem ocorrido.

As equações de fluxo de dados desta seção diferem em um aspecto das regras semânticas para os atributos no Capítulo 5. As dependências circulares entre os atributos não foram permitidas no Capítulo 5, mas vimos que as equações de fluxo de dados podem tê-las, como por exemplo *entrada*[S] e *saída*[S], que dependem um do outro em 10.6. Para o problema das definições incidentes, as equações de fluxo de dados podem ser resolvidas de modo a eliminar a circularidade — computam-se as equações não circulares da Fig. 10.21 para OUT e GEN. Uma vez que uma especificação não circular seja obtida, as técnicas do Capítulo 5 podem ser aplicadas de forma a se obter soluções eficientes para as equações de fluxo de dados.

Fluxo de Controle Geral

A análise de fluxo de dados precisa levar em conta todos os percursos de controle. Se os percursos de controle forem incertos a partir da sintaxe, as equações de fluxo de dados podem ser estabelecidas e resolvidas numa forma rígida pela sintaxe, como nesta seção. Quando os programas podem conter desvios goto, ou mesmo os mais disciplinados mecanismos de controle de fluxo, a abordagem sintática pode ser modificada para levar em conta os percursos de controle efetivos.

Várias abordagens podem ser adotadas. O método iterativo da próxima seção funciona para grafos de fluxo arbitrários. Ele é rápido para grafos de fluxo obtidos na presença de comandos *break* e *continue* isolados, tais construções podem ser tratadas sintaticamente usando-se métodos baseados em intervalos, que serão discutidos na Seção 10.8.

No entanto, a abordagem guiada pela sintaxe não precisa ser abandonada quando os enunciados *break* e *continue* forem permitidos

```
/* d1 */    i := m-1;
/* d2 */    j := n;
/* d3 */    a := u1;
            do
/* d4 */        i := i+1;
/* d5 */        j := j-1;
                if e3 then
/* d6 */            a := u2
                else begin
/* d7 */                a := u3;
                    break
                end
            while e2
```

Fig. 10.24. Programa contendo um enunciado `break`.

Após essa breve introdução, vamos considerar um exemplo que sugere como os enunciados *break* podem ser acomodados, de modo a estender o desenvolvimento de dados para a Seção 10.10.

Exemplo 10.14. O enunciado *break* contido no laço **do-while** da Fig. 10.24 é equivalente a um desvio para o final do laço. Como devemos definir o conjunto *gerados* para o seguinte enunciado?

```
do
    S1
    if E1 then
        S2
    else
        break
    S3
while E2
```

Do mesmo modo que *gerados* contém sempre $\{d_3\}$, onde d_3 é a definição de a em S_3, porque d_3 é a única definição gerada ao longo dos percursos de controle a partir do ponto de início até o de fim do enunciado. A definição d_2 estará em *gerados* somente quando todo o laço **do-while** for considerado.

Existe uma armadilha de programação que nos permite gerar o desvio causado pelo enunciado *break*, enquanto estivermos processando os enunciados dentro do corpo de um laço. Façamos os conjuntos *gerados* e *mortos* para um enunciado *break* conter, respectivamente, o conjunto vazio e o conjunto universal de todas as definições, como mostrado na Fig. 10.25. Os conjuntos remanescentes *gerados* e *mortos*, mostrados na Fig. 10.25, são determinados usando-se as equações de fluxo de dados da Fig. 10.21, com os conjuntos *gerados* mostrado acima do comando *mortos*. Os enunciados S_1 e S_2 representam uma seqüência de atribuições. Os conjuntos *gerados* e *mortos* são fáceis para ser determinados.

O ponto final de qualquer enunciado iterativo pode ser atingido de uma maneira ou de outra, de modo que não podemos fazer o conjunto *gerados* para o enunciado ser $\emptyset$ (vazio) e o conjunto *mortos* ser U (o resultado ainda seria uma estimativa conservativa correta, mas não ótima). Simplesmente o ponto final de um enunciado *if* pode somente ser atingido através da parte *then* ou os conjuntos *gerados* e *mortos* são **if**, na Fig. 10.25, são de fato os mesmos que aqueles do seu segundo filho.

Os conjuntos *gerados* e *mortos* ao nó **do** precisam levar em conta todos os percursos a partir do início até o final do enunciado **do**, de forma que são afetados pelo enunciado *break*. Vamos agora computar os conjuntos G e M iniciando de casos à medida que realizarmos uma travessia da parte contida na parte do enunciado *break*. A notação é que G_i e M_i representam as definições geradas e mortas à medida que o controle flui para o comando S_i, a partir do início do corpo do laço. O conjunto *gerados* para o enunciado **do** inteiro pode então ser determinado realizando-se a união de G_1 com um conjunto vazio para o corpo do laço, porque o controle pode atingir o final do laço quer pelo enunciado *break*, quer através do corpo do laço. Pela mesma razão, o conjunto *mortos* para o **do** é determinado tomando-se a interseção de M_1 e do conjunto *mortos* para o corpo do laço.

Encontramos antes de atingirmos o nó **if** temos $G = gerados(S_1 | S_2) = \{d_4, d_5\}$ e $M = mortos(S_1 | S_2) = \{d_1, d_2, d_3\}$. Ao ponto do nó **if**, estamos interessados no caso quando o controle flui para o comando *break* e dessa forma a parte *then* do comando condicional não possui efeito sobre G e M. O próximo nó ao longo do percurso ponto inicial é uma seqüência de atribuições, e, dessa forma, computamos os novos valores de G e M. Escrevendo S em lugar do enunciado representado pelo filho à esquerda do nó sequência daquele contendo d_4 e d_5, temos

$$G = gerados(S \mid \emptyset) \cup (G - mortos(S \mid \emptyset))$$
$$M = mortos(S \mid \emptyset) \cup (M - gerados(S \mid \emptyset)).$$

Por conseguinte, os valores de G e M ao atingirmos o enunciado *break* são $\{d_4, d_5\}$ e $\{d_1, d_2, d_3\}$, respectivamente. □

10.6 SOLUÇÃO ITERATIVA PARA AS EQUAÇÕES DE FLUXO DE DADOS

O conteúdo da última seção é simples e eficiente quando aplicável a coisas para linguagens como Fortran ou Pascal, que permitem grafos de fluxo arbitrários, mas é suficientemente geral. A Seção 10.10 discute a "caixa de intervalos" com, talvez, as melhores vantagens da abordagem dirigi-

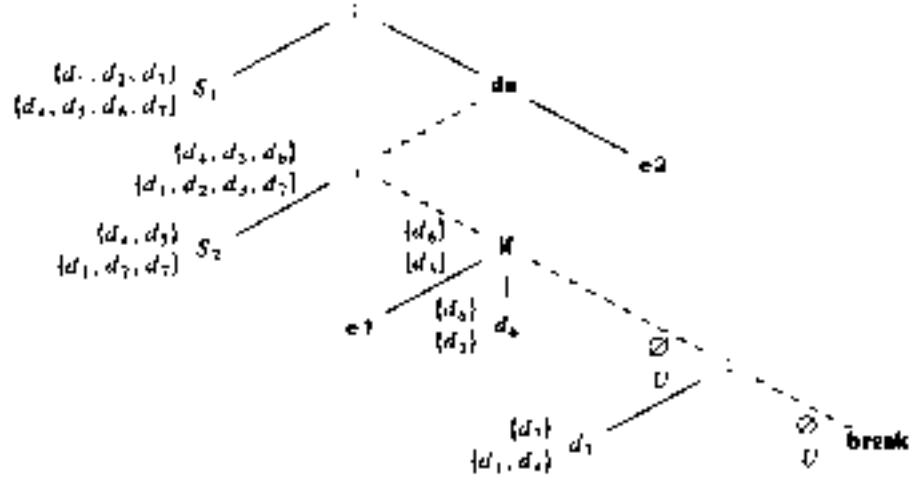


Fig. 10.25. Árvore de fluxo de dados mostrando os conjuntos *gerados* e *mortos*.

da pela saída para a análise de fluxo de dados sobre grafos de fluxo gerais e a necessidade de uma cuidadosa consideração de que informação

Agora vamos discutir uma outra importante abordagem à solução de problemas de fluxo de dados. Em vez de tentar usar a estrutura gramatical para dirigir o cômputo dos conjuntos *entrada* e *saída*, usaremos a estrutura do grafo de fluxo e uma técnica computacional para calcular *entrada* e *saída* diretamente. Veremos que este novo método nos dará uma oportunidade de apresentar um algoritmo para resolver os problemas de análise de fluxo de dados, mostrar alguns de suas aplicações e apontar as diferenças entre os problemas.

As equações para muitos problemas de fluxo de dados assumem a forma em que as informações estão sendo "geradas" e "consumidas". No entanto, existem duas importantes exceções a essas equações. Elas serão discutidas em detalhes.

1. As equações dadas nesta seção para as definições incidentes assumem que *para todo* B, $gerados[B]$ e $consumidos[B]$ são conjuntos de definições em termos de seus conjuntos *entrada*. Veremos também problemas que exigem que $gerados[B]$ e $consumidos[B]$ sejam computados em função de $entrada[B]$.

2. Quando definimos $entrada[B]$ como a união das definições que chegam ao início de B, estamos supondo que a ordem em que as definições chegam não importa. Em outras palavras, supomos que a informação $entrada$ é *conservativa*. Em contraste, quando definirmos problemas com expressões disponíveis, definiremos $entrada[B]$ como a interseção dos conjuntos correspondentes aos predecessores de B, porque uma expressão só estará disponível no início de B se estiver disponível ao final de cada predecessor de B. Na Seção 10.11 veremos exemplos de operadores de confluência.

Nesta seção veremos exemplos tanto de equações para as quais a informação é conservativa como para as quais não é. Veremos também exemplos de operadores de confluência.

Algoritmo Iterativo para Definições Incidentes

Para cada bloco básico B podemos definir $gerados[B]$, $consumidos[B]$ e $entrada[B]$ como na última seção, lembrando que cada bloco B pode ser visto como uma sequência de instruções, de modo que $gerados[B]$ e $consumidos[B]$ são definidos como antes. Assumindo que $gerados$ e $consumidos$ tenham sido computados para cada bloco, podemos escrever equações, mostradas em 10.9), abaixo, que relacionam $entrada$ e $saída$. O primeiro grupo de equações segue da observação que $entrada[B]$ é a união das definições que chegam de todos os predecessores de B. O segundo grupo é uma das equações que vimos sempre na forma geral (10.8), exceto que aqui usamos $saída$ para $gerados$ e $consumidos$.

$$entrada[B] = \bigcup_{P \text{ é um predecessor de } B} saída[P]$$

$$saída[B] = gerados[B] \cup (entrada[B] - consumidos[B]) \qquad (10.9)$$

Se um grafo de fluxo possui n blocos básicos, obtemos $2n$ equações a partir de (10.9). As $2n$ equações podem ser resolvidas de forma iterativa, como fizemos com os conjuntos *entrada* e *saída* na última seção, exceto que agora temos $2n$ equações em vez de uma equação para cada bloco básico. Para os ensinamentos **do-while**, começamos com a suposição de que todos os conjuntos são vazios e iteramos até que nenhum conjunto se altere. Aqui, começaremos com conjuntos vazios, uma vez que sabemos que $G(0)$ (que é o ponto de partida da iteração dos conjuntos $saída$) estarão vazios se os conjuntos $saída$ estiverem vazios. Em Cimplante estas variáveis são inicializadas com $gerados$ (Passo 10.1). Não necessitamos inicializar $entrada$ ou $saída$, pois as equações mais complexas não podem ser satisfeitas com conjuntos vazios.

Algoritmo 10.2 Definições incidentes.

Entrada. Um grafo de fluxo para o qual $consumidos[B]$ e $gerados[B]$ tenham sido computados para cada bloco B.

Saída. $entrada[B]$ e $saída[B]$, para cada bloco B.

Método. Usamos a variável booleana $mudou$ para controlar a iteração, que é executada repetidamente até que $mudou$ seja falso. Assumimos que $saída[B]$ é inicialmente $\emptyset$ para todos os conjuntos B e começamos a iteração com a suposição de que $entrada[B] = \emptyset$ para todos os blocos B. Na medida em que precisamos de uma variável temporária para acumular os membros de $entrada[B]$, vamos usar $saída[B]$ para esse propósito, atribuindo $entrada[B]$ a $saída[B]$ imediatamente antes de calcular $saída[B]$ a partir de $entrada[B]$. O algoritmo é delineado na Fig. 10.26.

Intuitivamente, o Algoritmo 10.2 propaga definições tanto quanto for possível sem que haja uma definição morta, isto é, uma definição que não aparece em nenhuma das possíveis execuções do programa. As noções precisas de "conservativo" e "definição morta" podem ser encontradas em referências mais adiantadas. Veremos provas formais da correção deste e de outros problemas de análise de fluxo de dados.

Podemos ver que o algoritmo termina eventualmente para todo $saída[B]$ finito, desde que em cada passo, para qualquer B, uma vez que uma definição seja adicionada a $saída[B]$ ela não é subsequentemente removida. Como o conjunto de todas as definições é finito, deverá haver um momento em que nenhuma nova definição é adicionada a nenhum conjunto $saída$, o qual *satisfaz* as equações $saída[B] = gerados[B] \cup (entrada[B] - consumidos[B])$ para todo B. Uma vez que isto ocorre, $mudou$ permanece **false**, e o algoritmo termina. Estamos certos em supor que os conjuntos $saída$ não mudarão, os conjuntos $entrada$ não mudarão na próxima passagem. E as equações entre

```
        Inicializa saída na suposição de que entrada[B] = ∅, para todos os B
(1)     para cada bloco básico B faça saída[B] := gerados[B];
(2)     mudou := true;                    /* variável booleana "pegajosa" */
(3)     enquanto mudou faça início
(4)         mudou := false;
(5)         para cada bloco básico B faça início
(6)             entrada[B] := ⋃ saída[P];
                              P é um
                              predecessor de B
(7)             saída_velha := saída[B];
(8)             saída[B] := gerados[B] ∪ (entrada[B] − consumidos[B]);
(9)             se saída[B] ≠ saída_velha então mudou := true
            fim
    fim
```

Fig. 10.26. Algoritmo para computar definições incidentes.

---

* Observe que não podemos garantir que o algoritmo termine se existir um ciclo no grafo de fluxo e uma definição gerar um valor para uma variável e não destruir nenhum valor para essa variável. Nesse caso, a definição pode circular indefinidamente.

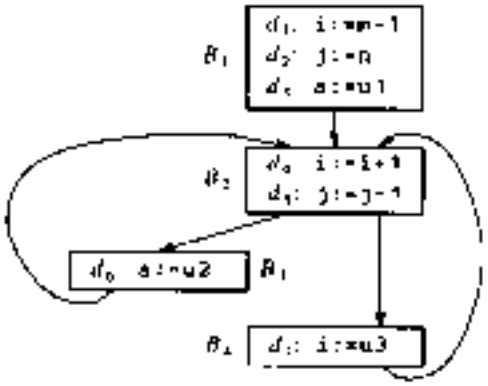

B_1
d_1: i := m − 1
d_2: j := n
d_3: a := u1
B_2
d_4: i := i + 1
d_5: j := j − 1
d_6: a := u2
B_3
B_4
d_7: i := u3
entrada [B1] = ∅
saída [B1] = {d1, d2, d3}
entrada [B2] = {d4, d5}
saída [B2] = {d2, d3, d4, d5}
entrada [B3] = {d6}
saída [B3] = {d5}
entrada [B4] = {d7}
saída [B4] = {d7, d4}

Fig. 10.27. Grafo de fluxo para ilustrar as definições incidentes.

são modificados, os conjuntos *saída* não poderão mudar de forma que, em todas as passagens subsequentes, não poderão haver mudanças.

Pode ser mostrado que um limite superior do número de vezes que o laço **enquanto** do algoritmo pode ser executado é igual à profundidade máxima de qualquer grafo de fluxo. Intuitivamente, o motivo é que, se uma definição atinge um determinado ponto, pode haver no máximo um caminho livre de ciclos, e o número de vezes que o laço **enquanto** pode ser executado é limitado superiormente pelo número de nós num percurso desse tipo. Ao se passar pelo laço uma vez, a definição propriamente dita pode ser incluída em algum dos percursos em questão.

De fato, se ordenarmos apropriadamente os blocos do laço **para** da linha (5) do Algoritmo 10.2, existem algoritmos de ordenação eficientes que fazem com que o número máximo de iterações num programa real esteja muito abaixo de 5 (ver a Seção 10.9). Como os conjuntos podem ser representados por vetores de bits e as operações sobre esses conjuntos podem ser implementadas através de operações lógicas sobre bits em vetores (ver Seção 10.2), o algoritmo é muito rápido na prática.

Exemplo 10.13. O grafo de fluxo da Fig. 10.27 foi derivado do programa de exemplo da Seção 10.1. Se aplicarmos o Algoritmo 10.2 a esse grafo de fluxo, uma passagem pelos blocos será suficiente. Vamos mostrar como as computações são feitas para o bloco B_3. Inicialmente, $entrada[B_3] = \emptyset$, e $saída[B_3]$ é igual a $gen[B_3] = \{d_6\}$. Quando a linha (3) é executada pela primeira vez, $entrada[B_3] = \{d_4, d_5\}$, pois $saída[B_2] = \{d_2, d_3, d_4, d_5\}$ e $saída[B_4] = \{d_7, d_4\}$. Assim, $saída[B_3] = \{d_4, d_5, d_6\}$. Na segunda passagem pelo bloco B_3, $entrada[B_3]$ não se altera, e $saída[B_3]$ também não se altera. Portanto, não são necessárias mais passagens pelo bloco B_3.

Em segundo lugar, consideremos $B = B_2$. Temos então

$$\begin{aligned} entrada[B] &= saída[B_1] \cup saída[B_3] \cup saída[B_4] \\ &= \{d_1, d_2, d_3, d_4, d_6, d_7\} = 1110111 \\ saída[B] &= gen[B] \cup (entrada[B] - matou[B]) \\ &= 0001100 \cup (1110111 - 0001101) = 0001100 \cup 1110010 = 1111110 \end{aligned}$$

Esse cômputo é ilustrado na Fig. 10.28. Ao final da primeira passagem, $saída[B_2] = 1111111$, refletindo o fato de que d_1 é gerado e que d_2, d_3 e d_4 atingem B_2 e não são mortos em B_2. A partir da segunda passagem não existem mudanças em quaisquer dos conjuntos *saída*, e essa forma o algoritmo termina.

Expressões Disponíveis

Uma expressão $x + y$ está *disponível* em um ponto p se cada caminho que vai do nó inicial até p avalia $x + y$ pelo menos uma vez e, após a última dessas avaliações antes de atingir p, nenhuma atribuição subsequente a x ou a y ocorre. Para as expressões disponíveis, dizemos que um bloco *mata* a expressão $x + y$ se ele atribuir a x ou a y e não recomputar $x + y$ subsequentemente. Um bloco gera a expressão $x + y$ se, independentemente de ele matar $x + y$ ou não, $x + y$ for computada pelo bloco subsequentemente à última atribuição a x ou a y.

Note-se que as noções de "matar" e "gerar" uma expressão disponível são estritamente iguais à noção das definições incidentes. Contudo, essas duas noções de "matar" e "gerar" estão separadas nessas duas aplicações de definições incidentes. Posteriormente, veremos como as computaremos eficientemente, como fizemos na Seção 10.5, desde que pudéssemos assumir modificações nos tipos em 6.2 (b) que envolvam apenas atribuições simples.

O uso primário das informações a respeito das expressões disponíveis está destinado à detecção das subexpressões comuns. Por exemplo, na Fig. 10.29, a expressão $x + 1$ no bloco B_3 será uma subexpressão comum se $x + 1$ estiver disponível no ponto p no início do bloco B_3. Essa disponibilidade não receberá uma atribuição de novo valor

Bloco B	Variável Inicial				Passagem 1				Passagem 2			
	entrada [B]		*saída* [B]		*entrada* [B]		*saída* [B]		*entrada* [B]		*saída* [B]	
B_1	0000	0000	1111	0010	0000	0000	1111	0010	0000	0000	1111	0010
B_2	0000	0000	1010	1100	1111	0010	0101	1110	0101	1110	1111	1110
B_3	0000	0000	1010	0010	0101	1110	0101	1110	0101	1110	1111	1110
B_4	0000	0000	1001	0001	0101	1110	1001	0001	1001	0001	1111	0001

Fig. 10.28. Cômputo de *entrada* e *saída*.

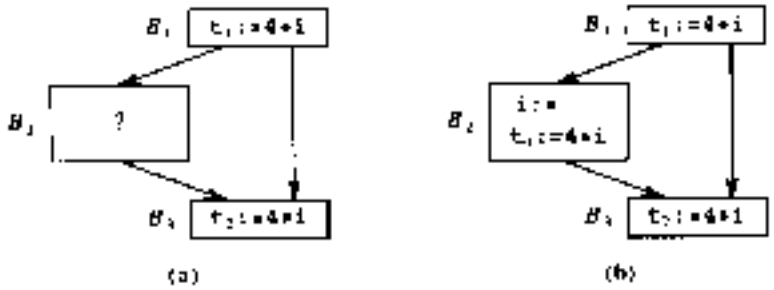


Fig. 10.29. Subexpressões comuns prévias e não-locais.

em um bloco B vale se, como na Fig. 10.29(a), $x+y$ for recomputado após ter recebido os valores finais em B.

Podemos facilmente computar o conjunto de expressões geradas por cada ponto num bloco, trabalhando a partir do início para o final do bloco. No ponto anterior do bloco, assumimos que não há expressões disponíveis. Se um ponto p tem um conjunto A de expressões disponíveis e a instrução em p gera uma expressão $x+y$ e x e y não são modificados após p no bloco, então $A \cup \{x+y\}$ é o conjunto de expressões disponíveis em p. Através dos dois sucessivos passos:

1. Adicionamos a A a expressão $x+y$.
2. Removemos de A qualquer expressão envolvendo x ou y.

Notemos que os dois passos precisam ser feitos na ordem correta, pois a instrução $x := x+y$ gera a expressão $x+y$, mas a mata após o final do passo. Vê-se o conjunto de expressões geradas para o Bloco B. O conjunto de expressões mortas são todas as expressões, digamos $y+z$, tais que y ou z sejam definidas no bloco e $y+z$ não seja gerada pelo mesmo.

Exemplo 10.16. Consideremos os quatro exemplos da Fig. 10.30. Após o primeiro, $x+i$ está disponível. Após o segundo, $x+i$ ainda está disponível, porém i não é definida. O terceiro objetiva $x+i$ disponível de novo, porque o valor de i é imediatamente modificado. Após o último exemplo, $x+i$ não está disponível, porque i foi modificada. Por conseguinte, nenhum enunciado gerador de todos os enunciados envolvendo i, [illegible] está morto.

Podemos encontrar expressões disponíveis numa forma que lembra a forma de definições incidentes sobre o ponto. Suponhamos que U seja o conjunto "universal" de todas as expressões que ocorrem no lado direito de um ou mais enunciados do programa. Para cada bloco B, seja $entrada[B]$ o conjunto de expressões em U que estejam disponíveis exatamente antes do início de B. Seja $saída[B]$ o mesmo para o ponto que se segue ao final de B. Definamos $entrada[B]$ como sendo $\emptyset$ se não houver predecessores de B, e definamos $saída[B]$ como sendo o conjunto de expressões em U mortas em B. Note-se que $entrada$, $saída$, e_gera e e_mata podem ser todos representados por vetores de bits. As seguintes equações relacionam os desconhecidos $entrada$ e $saída$, uns aos outros, e aos quantidades conhecidas e_gera e e_mata:

$$saída[B] = e\_gera[B] \cup (entrada[B] - e\_mata[B])$$

$$entrada[B] = \begin{cases} \emptyset & \text{para } B \text{ bloco inicial} \\ \bigcap_{P \text{ predecessor de } B} saída[P] & \text{caso contrário} \end{cases}$$

$$entrada[B_1] = \emptyset \text{ onde } B_1 \text{ é o bloco inicial} \qquad (10.16)$$

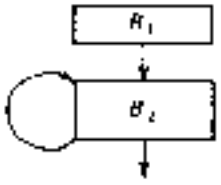
As equações (10.16) parecem quase idênticas às equações (10.9) para as definições incidentes. A principal diferença está em que entrada para o bloco inicial é tratada como um caso especial, pois é preciso saber sobre a base de que nada está disponível se o programa acabou de começar, ao invés de saber, mesmo que alguma expressão pudesse estar disponível ao longo de todos os caminhos até o início de qualquer parte do programa. Sendo $entrada[B_1] = \emptyset$, as soluções das equações corretamente deduzirão que certas expressões estão disponíveis antes do programa começar. A segunda e mais importante diferença é que o operador de combinação é a interseção, ao invés da união. Esse operador é o adequado porque uma expressão está disponível no início de um bloco somente se estiver disponível ao final de todos os seus predecessores. Em contraste, uma definição alcança o início de um bloco se alcançar o final de um ou mais de seus predecessores.

O uso de $\emptyset$ como valor de contorno nas equações (10.16) se comporta bem diferentemente daquelas de (10.9). Um pequeno teorema sobre conjuntos pessoais monótonos, para o qual o leitor é solicitado a encontrar uma demonstração, garante que existe uma única solução para as equações (10.16), e que qualquer algoritmo iterativo que corresponda à definição de "fim alcançado" e obtenha uma solução começando pela suposição de que nada alcança lugar algum é correto e termina. Na Seção 10.7, mostramos que uma definição d pode alcançar um ponto p a menos que exista um caminho de d a p que não passe por nenhum ponto que modifique a variável definida por d. Em contraste, para as equações (10.16), desejamos a maior solução possível, e devemos começar com uma aproximação que seja muito grande e trabalhar reduzindo.

Pode não ser óbvio que começando pela suposição de que tudo, exceto o conjunto $\emptyset$, está disponível em qualquer início e eliminando-se somente aquelas expressões para as quais possamos provar com certeza ao longo de qual não estarão disponíveis, possamos atingir um conjunto de expressões verdadeiramente disponíveis. Nesse caso as expressões disponíveis conservativo produzirão um subconjunto do conjunto exato de expressões disponíveis, e isso é o que temos de ter. O argumento é que todos os subconjuntos sejam corretos, o que é o nosso uso pretendido para as informações: é seguro substituir o cômputo de uma expressão disponível por um valor previamente computado. Veremos o Algoritmo 10.5 na próxima seção e descobrir que uma expressão está disponível só é simplesmente mais fácil de modificar o código.

Enunciados	Expressões Disponíveis
	nenhuma
	somente
	somente
	somente
	nenhuma

Fig. 10.30. Exemplos de expressões disponíveis.



$$O^{i+1} = G \cup (I^{i+1} - U)$$
$$I^{i+1} = \mathit{saída}(B_2) \cap O^{i+1}$$

$I^0 = \emptyset$	$I^0 = U$
$O^1 = G$	$O^1 = G \cup (U - \emptyset)$
$I^1 = \mathit{saída}(B_1) \cap G$	$I^1 = \mathit{saída}(B_1) \cap U$
$O^2 = G \cup (\mathit{saída}(B_1) - \emptyset)$	$O^2 = G \cup (\mathit{saída}(B_1) - U)$

Fig. 10.31. Duas formas de computar $\mathit{saída}(B_2)$ e possíveis valores.

Exemplo 10.17. Vamos nos concentrar no bloco B_2 na Fig. 10.31, para ilustrar o efeito da aproximação inicial de $\mathit{entrada}(B_2)$ ou $\mathit{saída}(B_1)$. Seguindo (10.11), as equações para $\mathit{entrada}(B_2)$ e $\mathit{saída}(B_1)$ respectivamente são. As equações de fluxo de dados para o bloco B_2 são,

$$\mathit{entrada}(B_2) = \mathit{saída}(B_1) \cap \mathit{saída}(B_2)$$
$$\mathit{saída}(B_2) = G_{B_2} \cup (\mathit{entrada}(B_2) - U_{B_2})$$

Essas equações foram reescritas como fórmulas de recorrência na Eq. (10.31), com I^i e O^i sendo as i-ésimas aproximações de $\mathit{entrada}(B_2)$ e $\mathit{saída}(B_1)$, respectivamente. A figura também mostra que, começando com $I = \emptyset$ obtemos $O^1 = G^1$. A computação que começa com $I = U$ obtemos as seguintes fórmulas para O^1. Também podemos ver que $\mathit{saída}(B_2)$ se iguala a O^i em cada caso porque as iterações convergem, exatamente para os valores mostrados.

Finalmente, a solução obtida começando com $I = U$ usando

$$\mathit{saída}(B_1) = G_1 \cup (\mathit{entrada}(B_1) - U_1)$$

Aqui, a exceção é $\mathit{saída}(B_1)$ que não são mantidas por B_1 estão disponíveis no final de B_1, exatamente como as expressões geradas por B_1 existem.

Algoritmo 10.3 (expressões disponíveis).

Entrada. Um grafo de fluxo, com $e\text{-}gera[B]$ e $e\text{-}mata[B]$ computados para cada bloco B. O bloco inicial é B.

Saída. O conjunto $\mathit{entrada}[B]$ para cada bloco B.

Método. Executar o algoritmo da Fig. 10.32. A explicação dos passos é similar àquela para a Fig. 10.26. □

Análise das Variáveis Vivas

Um conjunto de transformações que melhoram a qualidade do código depende de informações computadas na direção oposta ao fluxo de controle do programa. A menos que consideremos algumas de seguida. Nós estabelecidas variáveis vivas precisamos saber para cada ponto p se o valor de x em p poderá ser usado ao longo de algum percurso no grafo de fluxo começando por p. Se puder, dizemos que x está viva em p, caso contrário, x está morta em p.

Como vimos na Seção 9.7, uma importante aplicação para a informação sobre variáveis vivas surge durante a geração de código objeto. Após um valor ser computado num registrador e presumivelmente usado dentro de um bloco, não é necessário armazenar aquele valor se ele mesmo estiver morto no final do bloco. Igualmente, se todos os registradores estiverem ocupados e precisarmos de um outro registrador, devemos escrever o uso de um registrador com um valor morto, uma vez que aquele valor não tem que ser armazenado.

Vamos definir $\mathit{entrada}[B]$ como sendo o conjunto de variáveis vivas no ponto imediatamente antes do bloco B e definir $\mathit{saída}[B]$ como sendo o conjunto de variáveis vivas imediatamente após o bloco. Seja $\mathit{definido}[B]$ o conjunto de variáveis que recebam valores definidos anteriormente atribuídos em B antes de qualquer uso dessas variáveis em B e seja $\mathit{usado}[B]$ o conjunto de variáveis cujos valores possam ser usados em B antes de quaisquer definições dessas variáveis. Para consistência, as equações que relacionam $\mathit{definido}$ e usado com os conjuntos desconhecidos $\mathit{entrada}$ e $\mathit{saída}$ são

$$\mathit{entrada}[B] = \mathit{usado}[B] \cup (\mathit{saída}[B] - \mathit{definido}[B])$$
$$\mathit{saída}[B] = \bigcup_{S \text{ um sucessor de } B} \mathit{entrada}[S] \qquad (10.11)$$

O primeiro grupo de equações diz que uma variável está viva chegando a um bloco se for usada antes da redefinição dentro do bloco, ou se estiver saindo do bloco e não for redefinida dentro do mesmo. O segundo grupo de equações diz que uma variável está viva saindo de um bloco se e somente se estiver viva em algum de seus sucessores.

A relação entre (10.11) e as equações de definições mudantes (10.9) deve ser notada. Aqui, $\mathit{entrada}$ e $\mathit{saída}$ possuem papéis intercambiáveis, e usado e $\mathit{definido}$ substituem gerado e morto, respectivamente. Assim para (10.9), a solução para (10.11) não é necessariamente única e desejamos a menor solução possível. O algoritmo usado para a solução mínima é essencialmente uma versão de trás para a frente do Algoritmo 10.2. Como o mecanismo para detectar mudanças em cada um dos conjuntos $\mathit{entrada}$ é similar à forma como que

```
entrada[B] = ∅;
saída[B1] := gen[B1];  { entrada e saída do bloco inicial B1 }
para B ≠ B1 faça saída[B] = ∅ { saída inicial de todos os outros blocos };
mudou := true;
enquanto mudou faça início
    mudou := false;
    para B ≠ B1 faça início
        entrada[B] := ⋂_{P predecessor de B} saída[P];
        saída_anterior := saída[B];
        saída[B] := gen[B] ∪ (entrada[B] − mata[B]);
        se saída[B] ≠ saída_anterior então mudou := true
    end
end
```

Fig. 10.32. Cômputo das expressões disponíveis.

O leitor pode verificar que as mudanças nos conjuntos *saída* nos Algoritmos 10.2 e 10.3 preservam os detalhes para verificar a terminação do algoritmo.

Algoritmo 10.4 Análise das variáveis vivas.

Entrada. Um grafo de fluxo com os conjuntos *def* e *uso* de cada bloco computados para cada bloco.

Saída. *Saída*[*B*] é o conjunto de variáveis vivas à saída de cada bloco *B* do grafo de fluxo.

Método. Executar o programa da Fig. 10.33. □

Cadeias Definição-Uso

Um conjunto útil de informações derivadas da análise de variáveis vivas é o *encadeamento definição-uso* (ou encadeamento-du). Dizemos que uma variável é usada no comando *s* se seu valor *v* puder ser requerido. Por exemplo, todas as variáveis em cada um dos comandos $x := y + z$, $a := b + c$, $i := i + 1$ podem ser requeridas. O problema do encadeamento-du é o de computar para um ponto *p* o conjunto de usos *u* de uma variável *x*, digamos *x*, tal que exista um percurso de *p* até *u* que não redefina *x*.

Da mesma forma que com as variáveis vivas, se podemos computar *saída*[*B*], o conjunto de usos encontrados a partir do final do bloco *B*, podemos então computar as definições alcançadas a partir de qualquer ponto *p*, dentro do bloco *B*, esquadrinhando a porção de *B* que se segue a *p*. Em particular, se existir uma definição de uma variável *x* no bloco, podemos determinar a cadeia-du para aquela definição: a lista de todos os possíveis usos daquela definição. O método é análogo àquele descrito na Seção 10.5 para computar as cadeias ud e é deixado por conta do leitor.

As equações para computar as informações do encadeamento-du se parecem exatamente com (10.11), com a substituição de *definições entrando em B* no lugar de *uso*[*B*], mantendo o conjunto de usos *u* reconhecidos em *B* (ao invés do conjunto de pares (x, s) tais que *x* seja uma variável em *B* que possa ser usada em *s* e que nenhuma definição prevaleça em *B*. Em lugar de *definições*[*B*], considere-se o conjunto de pares (x, s) tais que *x* seja uma variável que se acha em *s*, não esteja em *B* e *B* tenha uma definição de *x*. Essas equações são resolvidas pelo análogo direto do Algoritmo 10.4 e mais uma vez abrigam a discussão sobre como tratar os elementos de vetores.

10.7 TRANSFORMAÇÕES DE MELHORIA DE CÓDIGO

Os algoritmos para realizar as transformações de melhoria de código introduzidas na Seção 10.2 recorrem a informações da análise de fluxo de dados. Nas duas últimas seções, vimos como essas informações podiam ser coletadas. Aqui consideramos a eliminação das subexpressões comuns, a propagação de cópias e as transformações para a movimentação de computações fazer invariantes para fora dos laços e a eliminação de variáveis de indução. Para muitas linguagens, melhorias significativas no tempo de execução podem ser atingidas melhorando o código dos laços. Quando as transformações são implementadas num compilador, é possível realizar algumas das transformações sugeridas somente. No entanto, vamos apresentar as idéias subjacentes às transformações independentemente.

A ênfase desta seção será nas transformações globais que usam informações a respeito de programa como um todo. Como vimos nas duas últimas seções, a análise global do fluxo de dados não consegue usualmente pontos dentro dos blocos básicos. As transformações globais, portanto, são, por conseguinte, um substituto para as transformações locais, muitas precisam ser realizadas. Por exemplo, a localização e propagação global de subexpressões comuns é raramente suficiente preocupados com o fato de uma expressão ser gerada por um bloco e não se apresentar reconhecida diversas vezes dentro do mesmo.

Eliminação Global das Subexpressões Comuns

O problema de fluxo de dados das expressões disponíveis, discutido na última seção, nos permite determinar se uma expressão como $a + b$ no ponto *p* no grafo de fluxo é uma subexpressão comum. O algoritmo seguinte formaliza as idéias intuitivas apresentadas na Seção 10.2 para eliminar as subexpressões comuns.

Algoritmo 10.5 Eliminação global das subexpressões comuns.

Entrada. Um grafo de fluxo com informações a respeito das expressões disponíveis.

Saída. Um grafo de fluxo revisado.

Método. Para cada comando da forma $x := y + z$ [1] que esteja disponível ao início do bloco de que faz parte e para o qual y e z não estejam definidos antes do comando $x := y + z$ naquele bloco, fazer o seguinte:

1. Para descobrir as avaliações de $y + z$ que alcançam o bloco de $x := y + z$, seguimos os arcos do grafo de fluxo para trás a partir do bloco de $x := y + z$. No entanto, não seguimos através de qualquer bloco que avalie $y + z$. A última avaliação de $y + z$ em cada bloco encontrado é uma avaliação de $y + z$ que alcança $x := y + z$.

2. Criar uma nova variável u.

3. Substituir cada comando $w := y + z$ encontrado em 1 por

$$\begin{aligned} u &:= y + z \\ w &:= u \end{aligned}$$

4. Substituir o comando $x := y + z$ por $x := u$. □

Alguns comentários sobre este algoritmo estão em ordem.

1. A busca no passo 1 do algoritmo, para avaliações de $y + z$ que alcançam o comando $x := y + z$, pode também ser formulada como um problema de análise de fluxo de dados. No entanto, não há necessidade de resolvê-lo para todas as expressões $y + z$ e todos os comandos ou blocos, porque muita dessa informação não será coletada. Antes, devemos realizar uma busca sobre o grafo de fluxo para cada comando $x := y + z$ relevante.

2. Nem todas as modificações realizadas pelo Algoritmo 10.5 são sempre benéficas. Podemos provavelmente desejar limitar em 1 o número de diferentes avaliações encontradas no passo 1, que alcançam $x := y + z$. No entanto, a propagação de cópias, a ser discutida em seguida, permite frequentemente que o benefício seja obtido mesmo quando várias avaliações de $y + z$ alcançam $x := y + z$.

[1] Ou qualquer outro comando que tenha a forma geral $x := y + z$.

3. O Algoritmo 10.5 não omitirá o fato de que $x + y$ e $a + y$ possuem o mesmo valor.

$$x = x + 1 \qquad\qquad a = x + 1$$
$$y = a + y \quad \text{VS} \quad y = a + y$$

Porém essa distinção pode ser simples para as subexpressões comuns, considerando somente as expressões literais em si ou através dos valores computados pelas expressões. Knoop, Rüthing e Steffen [1994] apresentam um método para capturar tais equivalências em uma passagem, discutimos essas idéias na Seção 10.6.1. No entanto, essas equivalências podem ser capturadas através de múltiplas passagens do Algoritmo 10.5 e deve-se considerar repetir a otimização quando encontrar mais pontos de melhoria. Se x e a são variáveis temporárias que não são usadas fora do bloco no qual figuram, a substituição pela expressão $x + y$ pode ser capturada tratando-se as expressões de forma especial, como no próximo exemplo.

Exemplo 10.18. Vamos supor que não existam atribuições a a ou x no grafo de fluxo da Fig. 10.34(a), de forma a que possamos dizer seguramente que $a + y$ e $x + y$ sejam subexpressões comuns. O problema é determinar essas subexpressões comuns.

A subexpressão comum $a + y$ na Fig. 10.34(a) foi eliminada na Fig. 10.34(b). Uma forma de determinar que $a + y$ e $x + y$ são subexpressões comuns é substituir a por x usando a propagação de cópias, a ser discutida em seguida, ambas as expressões se tornam $x + y$, que pode ser eliminada usando o Algoritmo 10.5. Notemos que a mesma nova variável t_5 é inserida em ambos os blocos da Fig. 10.34(b), de forma que a propagação local de cópias funcionará para converter ambos $t_4 = a + y$ e $t_5 = x + y$ em $t_5 = t_3$.

Existe uma outra forma que leva em conta o fato das variáveis temporárias serem inseridas pelo compilador e usadas somente dentro dos blocos em que ocorrem. Vemos a seguir mais detalhadamente a forma através da qual as expressões são representadas durante a compilação. As expressões idênticas possuem todas o mesmo número de valor, e a idéia é atribuir um número a cada expressão, usar um valor de *hash* com a *lexema* representando a expressão de número. As técnicas dos números de valor da Seção 9.2 podem ser aplicadas durante a numeração de expressões, de forma a tratar as variáveis temporárias de forma especial.

Mais detalhadamente, suponhamos que $t_1 = 4 * i$ tenha o número de valor 15. As expressões $t_2 = t_1 + t_2$ não obtêm o mesmo número de valor se os números de valor de t_1 e t_2 não forem também números de valor. Suponhamos que o número de valor resultante seja 16. Por conseguinte, $t_3 = t_2 + 18$ não representa também $a[t_2]$, quanto $t_4 = a[t_3]$, durante a análise de fluxo de dados e podemos determinar que $t_4 = a[t_3]$ exija dispensável e possa ser eliminada. O código resultante é indicado na Fig. 10.34(c), com somente (15) e (18) para representar as variáveis temporárias correspondentes às expressões com seus números de valor. Efetivamente, t_4 é inútil e deveria ser eliminada, de forma que a propagação de variáveis vivas igualmente. Como t_5 sendo uma variável temporária, não deveria ser computada em B_4, pois todos os usos de t_5 deveriam ser substituídos por usos de t_3.

Propagação de Cópias

O Algoritmo 10.5 recém-apresentado e vários outros algoritmos tais como a eliminação de variáveis de indução, discutidos mais tarde nesta seção, introduzem enunciados de cópia da forma $x = y$. As cópias também podem ser geradas diretamente pelo gerador de código intermediário, apesar da natureza delas envolver variáveis temporárias locais a um bloco, e poderem ser removidas pela construção de GDAs discutida na Seção 9.8. É possível algumas vezes eliminar o enunciado de cópia $x = y$ se determinarmos todos os locais onde essa definição de x é usada. Podemos, então, substituir x por y e todas esses locais uma vez procedimentado que as seguintes condições sejam satisfeitas para cada tal uso de x.

1. O enunciado c precisa ser a única definição de x que alcança esse uso, o que é satisfeito para todos os usos somente de x.

2. A única percurso de x para o qual todos os percursos que alcançam c passam por c uma ou mais vezes, mas não passam por c uma segunda vez, não existam atribuições a y.

A condição 1 pode ser verificada usando-se as informações do cálculo-definição-uso, e a 2 usando a relação $I24$. Estabelecemos um novo problema de análise de fluxo de dados. Para cada enunciado d é o conjunto de cópias $x = y$ tais que cada percurso de x para y contém d. Em particular, até o início de B contém o enunciado d e subsequentemente nenhuma atribuição a y ocorre, de forma que não existam atribuições a y. O conjunto $saída[B]$ pode ser correspondentemente definido, com respeito, porém, ao final de B. Observemos que o enunciado de cópia $x = y$ é gerado no bloco B se ocorrer em B e não existir atribuição subsequente a x dentro de B. Uma cópia que entra em B e não sofre atribuição em B é dita ser preservada em B. A noção de que as atribuições a x "matam" $x = y$ é tratada como de costume, exceto que, de novo, as atribuições a x também fazem isso especial neste problema. Notemos a importante consequência do fato de que diferentes atribuições a x podem matar uma das outras: $entrada[B]$ pode conter somente um enunciado de cópia com x à esquerda.

Seja U o conjunto "universal" de todos os enunciados de cópia no programa. É importante notar-se que os diferentes enunciados em U são diferentes em U. Definimos $gerados[B]$ como o subconjunto de todas as cópias geradas no bloco B e $mortos[B]$ como sendo o conjunto de cópias em U que sejam mortas em B. As seguintes equações relacionam as quantidades definidas

$$saída[B] = gerados[B] \cup (entrada[B] - mortos[B])$$
$$entrada[B] = \bigcap_{\text{predecessor } P \text{ de } B} saída[P] \text{ para } B \text{ não inicial} \qquad (10.21)$$
$$entrada[B_1] = \emptyset \text{ onde } B_1 \text{ é o bloco inicial}$$

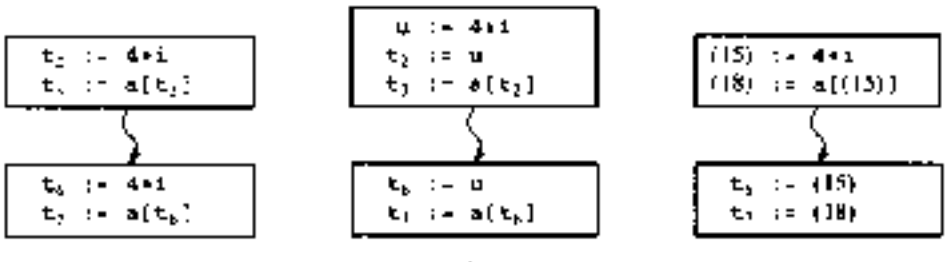


Fig. 10.34. Eliminação de subexpressão comum [...]

As equações 10.12 serão idênticas às equações 10.10, se substituirmos todas as ocorrências de *gerado* por *preservado*. Por conseguinte, as equações 10.12 podem ser resolvidas pelo Algoritmo 10.3 sem nenhuma modificação. Exploramos mais esse assunto fornecendo, no exemplo a seguir, um exemplo que explicita algumas das nuances da otimização de cópias.

Exemplo 10.14. Considere novamente o grafo de fluxo da Fig. 10.35. Aqui, $preservado|B_1| = \{x := y, x := z\}$, $preservado|B_2| = \{x := y\}$, $preservado|B_3| = \{x := z\}$, $preservado|B_4| = \{x := y, t := x\}$ e $preservado|B_5| = \{x := z, t := x\}$. Também, $entrada|B_1| = \emptyset$, pois x recebe uma atribuição em B_1, independente de $saída|B_5|$, $saída|B_2| = \{x := y\}$, pois x recebe uma atribuição em B_1 e $saída|B_3| = \{x := z\}$, pela mesma razão.

Os demais conjuntos $gerado$ e $morto$ são $\emptyset$. Igualmente, $entrada|B_1| = \emptyset$ pelas equações 10.12. O Algoritmo 10.3 em uma passagem determina que

$$entrada|B_2| = entrada|B_3| = saída|B_1| = \{x := y, x := z\}$$

Analogamente, $saída|B_1| = \emptyset$:

$$saída|B_4| = entrada|B_5| = saída|B_2| \cup saída|B_3| = \{x := y, x := z\}$$

Finalmente, $entrada|B_4| = saída|B_5| = saída|B_2| \cap saída|B_3| = \{x := y, x := z\}$.

Observamos que nenhuma das duas cópias $x := y$ e $x := z$ "chega ao topo do laço", em B_4, nos termos do Algoritmo 10.5. Isto, porém, ainda que não evidencie que essas duas definições de "atribuem" B_4, dentro da significação das definições que estudamos. Por conseguinte, nenhuma cópia pode ser propagada na medida em que não é possível substituir x por y, respectivamente, z em todos os usos de x cuja definição é $x := y$, respectivamente, $x := z$ chega. Poderíamos substituir x por y, ou z, em B_4, mas essa não é uma melhoria de código.

Algoritmo 10.6 Propagação de cópias.

Entrada. Um grafo de fluxo G com tabelas-ud bem construídas e as definições incidentes ao bloco B de cada $entrada|B|$ representando a solução para esse item do Algoritmo 10.3, isto é, o conjunto de cópias $x := y$ que chegam ao bloco B ao longo de cada percurso, sem atribuições a x ou a y, após a última ocorrência de $x := y$ no percurso. Precisamos também de tabelas-ud bem construídas para cada definição.

Saída. Um grafo de fluxo reescrito.

Método. Para cada cópia $x := y$, faça o seguinte:

1. Determine aquelas usos de x que são atingidos por esta definição, isto é, nominalmente, $x := y$.

2. Determine se para cada uso de x encontrado em (1), $x := y$ está em $entrada|B|$, onde B é o bloco desse uso particular, e também se nenhuma definição de y onde $x := y$ ocorre antes desse uso de x dentro de B. Relembre que se $x := y$ está em $entrada|B|$, então $x := y$ é a única definição de x que atinge B.

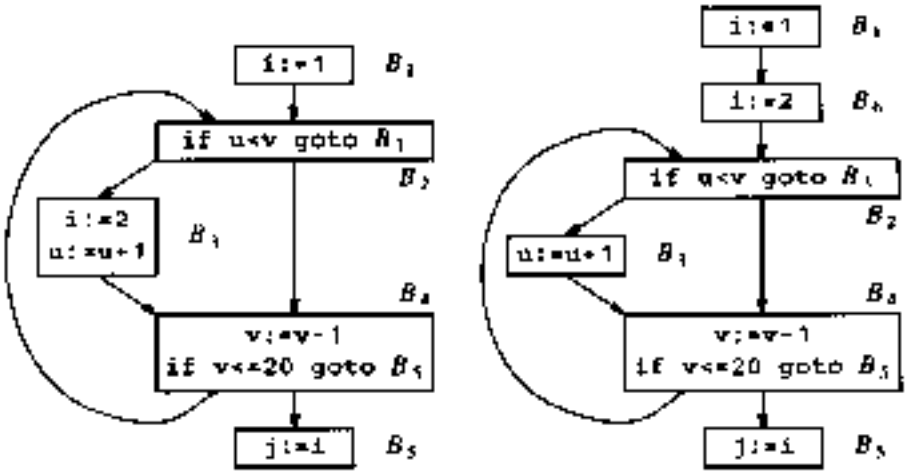
3. Se a resposta às verificações de (2) é sim, remova $x := y$ e substitua por todos os usos de x encontrados em (1) y.

Detecção de Cômputos Laço-Invariantes

Podemos fazer uso das cadeias-ud para detectar as computações que geram laço-invariantes, isto é, aquelas cujos valores não se modificam a cada laço que o contorno executa. O laço G consiste em um conjunto de blocos com a propriedade que dentre todos os outros blocos, existe um único bloco, de entrada, através do qual o controle entra no laço. Também existe pelo menos um laço tal que, pelo menos, uma forma de voltar ao cabeçalho, a partir de qualquer bloco do mesmo.

Se uma atribuição $x := \ldots$ existe em uma posição do laço, mas todas as possíveis definições de y em $\ldots := y$ estejam fora do laço, mas chegam até o início do laço onde $\ldots := y$ é executado, então $\ldots := y$ é, por conseguinte, laço-invariante, porque seu valor será o mesmo a cada vez que $\ldots := y$ for encontrado por algum tempo quando o controle estiver dentro do laço. Todas essas atribuições podem ser detectadas a partir das cadeias-ud, isto é, a lista de todos os pontos de definição de y que atingem a atribuição $\ldots := y$.

Tendo reconhecido que o valor de $\ldots := y$ computado em $x := y$ não se modifica dentro do laço, suponhamos inicialmente que esta atribuição seja $x := c$, onde c poderia somente ser definido fora do laço. Nesse caso, $x := c$ também é laço-invariante. Podemos usar as cadeias-ud para localizar repetidas passagens sobre um laço, descobrindo cada vez mais computações cujos valores sejam laço-invariantes. Se tivermos também as cadeias-ud quanto às cadeias-du, não precisaremos nem mesmo de fazer passagens repetidas sobre o código. A cadeia-du para a definição $x := \ldots$ lista-nos diretamente todos os usos de x que podem ser afetados por $x := \ldots$. Assim, se soubermos que $x := \ldots$ é laço-invariante, podemos imediatamente concluir que todos os usos de x listados na cadeia-du são também laço-invariantes. Podemos, portanto, aplicar a mesma regra para localizar repetidas passagens sobre um laço, descobrindo cada vez mais computações cujos valores sejam laço-invariantes. Se tivermos também as cadeias-du quanto às cadeias-ud, não precisaremos nem mesmo de fazer passagens repetidas sobre o código. A cadeia-ud para a definição $x := \ldots$ lista-nos diretamente todas as definições que podem ter afetado o valor de x usado em $\ldots := x$. Assim, se soubermos que todas as definições de x que podem ter afetado o valor de x usado em $\ldots := x$ estão fora do laço G, ou são laço-invariantes, podemos concluir que $\ldots := x$ também é laço-invariante. Podemos usar as cadeias-ud repetidamente



(a) Antes (b) Depois

Fig. 10.36. Exemplo de movimentação de código ilegal.

Realizando a Movimentação de Código

Tendo encontrado os comandos invariantes de laço de um laço, podemos aplicar algoritmos de otimização de código que movem esses comandos para o pré-cabeçalho do laço. As três seguintes condições asseguram que a movimentação de código não modifique o que o programa computa. Pode haver outras condições e absolutamente essencial. As mesmas foram selecionadas porque são fáceis de verificar e aplicar a situações que ocorrem em programas reais. Iremos discutir mais uma possibilidade de relaxar essas condições.

As condições para mover o comando invariante s: $x := y + z$ são:

1. Um bloco contendo s deve dominar todos os nós de saída do laço. Onde uma saída de um laço é um nó com um sucessor fora do laço.

2. Não há outro comando no laço que atribua valor a x. De novo, se x for uma variável temporária, tal atribuição somente uma vez, esta condição é certamente satisfeita e não precisa ser verificada.

3. Nenhum uso de x no laço é atingido por qualquer definição que não s. Esta condição também será satisfeita naturalmente, se x for uma variável temporária.

Os três próximos exemplos ilustrarão as condições acima.

Exemplo 10.20. Vamos mover um comando que nos parece ser executado dentro de um laço, porém posição fora do mesmo pode mudar o que um programa computa. Como mostramos na Fig. 10.36. Esta observação motiva a condição (1) uma vez que sabemos todo que comandos em saídas não podem deixar de ser executados assumindo que o laço não roda para sempre.

Consideremos o grafo de fluxo mostrado na Fig. 10.36(a). B_2, B_3 e B_4 formam um laço com cabeçalho B_2. A comando $i := 2$ em B_3 é claramente invariante. No entanto, B_3 não domina B_5, a única saída do laço. Se movermos $i := 2$ para pré-cabeçalho, independentemente do comando B_1, como mostrado na Fig. 10.36(b), podemos multiplicar o valor atribuído a u em B_3, naqueles casos em que B_3 jamais seria executado. Por exemplo, se $u = v$ inicialmente, quando B_1 for atingido pela primeira vez, a Fig. 10.36(a) terá i igual a u em B_5, porque B_3 jamais será atingido, enquanto que a Fig. 10.36(b) terá i igual [illegible]

Exemplo 10.21. A condição (2) é especialmente importante para instruções dentro de um laço. Por exemplo, a estrutura do grafo de fluxo na Fig. 10.37 é a mesma que aquela da Fig. 10.36(a) e temos a opção de mover $i := 2$ para o pré-cabeçalho B_1, como na Fig. 10.36(b).

Como B_1 na Fig. 10.37 domina a saída B_5, a condição (1) não impede que $i := 2$ seja movido para o pré-cabeçalho B_1. No entanto se fizermos, iremos atribuir i a 2 sempre que B_1 for executado e teremos valor i quando atingirmos B_5 mesmo se saltarmos para sequência

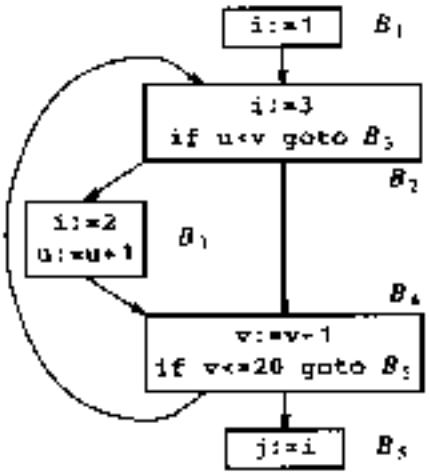


Fig. 10.37. Ilustrando a condição (2).

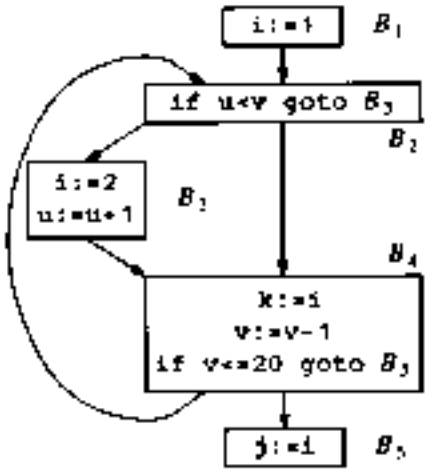


Fig. 10.38. Ilustrando a condição (3).

seqüência $B_1 \to B_2 \to B_3 \to B_2 \to B_3 \to B_4$. Por exemplo, consideremos o que acontece se o valor de i for ... antes de ... quando B_3 for atingido pela primeira vez. Se o enunciado ... estiver em B_1, faremos ... igual a ... em B_4, mas se ... for removido para o pré-cabeçalho, faremos ... igual a ...

Exemplo 10.22. Vamos agora considerar a Fig. 10.39 (c), ... que bloco B_4 da Fig. 10.38 é atingido por ... no bloco B_2, bem como por ... em B_3. Conseqüentemente, não poderíamos mover ... para o pré-cabeçalho, porque o valor de ... atingindo sobre B mudaria no caso de ... Por exemplo, se ... então ... teria igual ... no grafo de fluxo da Fig. 10.38, mas se ... for movido para o pré-cabeçalho, fazemos ... igual a ... uma vez e para todas as situações.

Algoritmo 10.8 Movimentação de código.

Entrada. Um laço L com informações do encadeamento ud e informações a respeito de dominadores.

Saída. Uma versão revisada do laço com um pré-cabeçalho e (possivelmente) alguns enunciados movidos para o pré-cabeçalho.

Método.

1. Usar o Algoritmo 10.7 para encontrar os enunciados laço-invariantes.
2. Para cada enunciado s definindo x encontrado no passo (1), verificar:

   i) se está num bloco que domine todas as saídas de L,
   ii) se x não é definido em algum local de L e
   iii) se todos os usos em L de x podem somente ser atingidos pela definição de x no enunciado s.

3. Mover, na ordem encontrada pelo Algoritmo 10.7, cada enunciado s encontrado em (1) e atendendo às condições (2i), (2ii) e (2iii), para o novo pré-cabeçalho criado, providenciando que quaisquer operandos de s que sejam definidos no laço L (no caso de s ter sido encontrado no passo (2) do Algoritmo 10.7) tenham tido seus enunciados de definição previamente movidos para o pré-cabeçalho.

Para compreender por que nenhuma mudança para o que o programa computa pode ocorrer, as condições (2i) e (2ii) do Algoritmo 10.8 asseguram que o valor de x computado em s precisa ser o valor de x após qualquer bloco de saída de L. Quando movemos s para o pré-cabeçalho, s ainda será a definição de x que atinge o final de qualquer bloco de saída de L. A condição (2iii) assegura que quaisquer usos de x dentro de L usavam, e irão continuar a usar, o valor de x computado por s.

Para ver por que a transformação não pode aumentar o tempo de execução do programa, temos apenas que observar que a condição (2i) assegura que s seja executado pelo menos uma vez cada vez que o controle entrar em L. Após a movimentação de código, o mesmo será executado exatamente uma vez no pré-cabeçalho e não mais em L sempre que o controle lá entrar.

Estratégias Alternativas de Movimentação de Código

Podemos relaxar a condição (1) em alguma extensão se estivermos dispostos a correr o risco de efetivamente aumentarmos um pouco o tempo de execução do programa; naturalmente, jamais mudaremos o que o programa computa. A versão menos restrita da condição (1) da movimentação de código [isto é, o item (2i) no Algoritmo 10.8] permite que movamos um enunciado s atribuindo x somente se:

1'. O bloco que contém s ou domina todas as saídas do laço ou x não é usado fora do laço. Por exemplo, se x for uma variável temporária, podemos estar certos (em muitos compiladores) de que o valor será usado somente em seu próprio bloco. Em geral, a análise de variáveis vivas é necessária para informar se x está vivo em qualquer saída do laço.

Se o Algoritmo 10.8 for modificado para usar a condição (1'), o tempo de execução irá ocasionalmente aumentar ligeiramente, mas podemos esperar que nos saiamos razoavelmente bem na média. O algoritmo modificado deve mover para o pré-cabeçalho certas computações que poderiam não ser executadas dentro do laço. Isto não somente arrisca retardar o programa significativamente, mas também pode causar um erro em certas circunstâncias. Por exemplo, a avaliação de uma divisão x/y num laço pode ser precedida por um teste para ver se $y = 0$. Se movermos x/y para o pré-cabeçalho, uma divisão por zero pode ocorrer. Por essa razão, não é inteligente usar a condição (1') a menos que a otimização possa ser inibida pelo programador, ou que apliquemos a condição mais restrita aos enunciados de divisão.

Mesmo que nenhuma das condições (2i), (2ii) e (2iii) do Algoritmo 10.8 seja atendida por uma atribuição $x := y+z$, podemos ainda realizar a computação $y+z$ fora do laço. Criamos uma nova variável temporária t e fazemos $t := y+z$ no pré-cabeçalho. Em seguida, substituímos $x := y+z$ por $x := t$ dentro do laço. Em muitos casos podemos, em seguida, propagar o enunciado de cópia $x := t$, como discutido antes nesta seção. Notemos que, se a condição (2iii) do Algoritmo 10.8 for atendida, isto é, se todos os usos de x no laço L forem definidos em $x := y+z$ (agora $x := t$), pode-se remover o enunciado $x := t$ substituindo os usos de x em L por usos de t e colocando $x := t$ após cada saída do laço.

A Manutenção das Informações de Fluxo de Dados após a Movimentação de Código

As transformações do Algoritmo 10.8 não mudam as informações de encadeamento ud, uma vez que, pelas condições (2i), (2ii) e (2iii), todos os usos da variável que recebem a atribuição através de um enunciado movido s e que eram atingidos por s ainda o são a partir de sua nova posição. As definições das variáveis usadas por s ou estão fora de L, caso em que atingem o pré-cabeçalho, ou estão dentro de L, caso em que, pelo passo (3), serão movidas para o pré-cabeçalho à frente (isto é, antes) de s.

Se as cadeias ud são representadas por listas de apontadores para apontadores de enunciados (ao invés de através de listas de apontadores para enunciados), podemos manter as cadeias ud quando movermos o enunciado s simplesmente modificando o apontador para s. Isto é, criamos para cada enunciado s um apontador p_s, o qual sempre aponta para s. Colocamos p_s em cada cadeia ud que contenha s. Como conseqüência, não importa a quanto movermos s, teremos apenas que modificar p_s, independentemente de quantas cadeias ud s estiverem participando. Naturalmente, esse nível extra de indireção custa algum tempo e espaço do compilador.

Se representarmos as cadeias ud por uma lista de endereços de enunciados (apontadores para enunciados), podemos ainda manter as cadeias ud à medida que movimentarmos os enunciados. Mas necessitaremos, então, de cadeias du por uma questão de eficiência. Quando movermos s, percorreremos a sua cadeia du, mudando a cadeia ud em todos os usos que se referem a s.

As informações sobre o dominador são modificadas ligeiramente pela movimentação de código. O pré-cabeçalho é agora o dominador imediato do cabeçalho e o dominador imediato do pré-cabeçalho é o nó que anteriormente era o dominador do cabeçalho. Isto é, o pré-cabeçalho é inserido na árvore dos dominadores como pai do cabeçalho.

Eliminação das Variáveis de Indução

Uma variável v é chamada de uma *variável de indução* de um laço L, se cada vez que a instrução destino de valor v for encontrada em dentro de L, v for incrementada por alguma constante. Frequentemente, uma variável de indução é incrementada pela mesma constante cada vez que o controle passa pelo laço de volta ao início. Uma variável básica de indução encabeçada por i é i ou i. Se encontrarmos nossos testes ligados com variáveis que são incrementadas ou decrementadas de uma constante diferente a cada vez que a variável que controla o laço é incrementada, podemos ajustar o conjunto de constantes relacionadas com a variável de indução pode muitas vezes diferir em iterações diferentes.

Uma situação comum é aquela em que uma variável de indução, digamos i, é incrementada ou decrementada por alguma outra variável de indução, digamos j, imediatamente após uma atribuição $i = j$. Se j for decrementada em paralelo com i, então i será sempre igual a j mais alguma constante. Frequentemente, j não será usada para nada, exceto para incrementar i. Podemos então eliminar j e substituir todos os testes de j por testes sobre i.

Os algoritmos que se seguem lidam com uma classe restrita de variáveis de indução para simplificar a apresentação. Algumas extensões dos algoritmos podem ser feitas alternando-se mais casos, mas outras requerem que sejam providas estruturas adequadas para as propriedades envolvidas nos cálculos aritméticos usados.

Iremos procurar pelas *famílias básicas de indução*, que são aquelas variáveis cujos únicos incrementos dentro do laço L são da forma $i = i + c$, onde c é uma constante. Procuraremos, a seguir, por variáveis de indução adicionais j que sejam de tal modo que sempre que j for definida dentro do laço por alguma atribuição $j = e$, e for uma função linear de alguma variável básica de indução i. Dizemos que j pertence à *família de* i.

Algoritmo 10.9 Detecção de variáveis de indução.

Entrada. Um laço L com informações de definições-alcançadas e de compilação-inovante, previamente do Algoritmo 10.7.

Saída. Um conjunto de variáveis de indução. Associada a cada variável de indução está uma tripla (i, c, d), onde i é uma variável básica de indução, c é uma constante e d é uma constante ou indefinido. Para uma variável básica de indução i, $c = 0$ e $d = 1$. Se j pertence à família de i, então $j = ci + d$ no ponto onde j é definida. Dizemos que j pertence à *família de* i. A variável básica de indução i pertence à sua própria família.

Método.

1. Procurar todas as variáveis básicas de indução, examinando cada uma das instruções de L. Usar as informações de compilação-inovante. Associada a cada variável básica de indução está a tripla $(i, 1, 0)$.

2. Procurar pelas variáveis k com uma única atribuição $k = e$ dentro de L, tendo uma das seguintes formas:

$$k = j * b, \quad k = b * j, \quad k = j / b, \quad k = j \pm b, \quad k = b \pm j$$

onde b é uma constante e j uma variável de indução básica ou não.

Se j for básica, então k está na família de j. A tripla para k depende da instrução que a define. Por exemplo, se $k = j * b$, então a tripla para k é $(b, 0)$. As triplas para as outras quatro expressões podem ser determinadas similarmente.

Se j não é básica, tomar i da família de j. Novos exigem as condições são que

a) j seja a única variável de indução a ser definida entre o único ponto de atribuição a k e k e

b) Sempre que for definida dentro de L, a forma de k seja $ci + d$.

O caso mais fácil será quando k for definida em uma instrução que tenha j no mesmo bloco, caso que é fácil de se verificar. Em geral, as informações sobre definições-alcançadas nos permitirão verificação que necessitamos se analisarmos o grafo de fluxo do laço L para determinar os nós. Iremos, por conseguinte, aquelas definições dos percursos entre a atribuição a j e a atribuição a k.

Computadores a tripla para k a partir da tripla (i, c, d) para j e a instrução que define k. Por exemplo, se definirmos $k = bj$, então k é (i, bc, bd). Note-se que as multiplicações em bc e bd podem ser feitas à medida que a análise prossegue, porque b, c e d são constantes.

3. Uma vez que as famílias de variáveis de indução tenham sido encontradas, modificar as instruções para computar uma variável de indução de forma que as adições ou subtrações e a vez de multiplicações. A substituição dessa instrução mais dispendiosa por uma mais econômica é chamada de *redução de força*.

Exemplo 10.23. O laço que consiste no bloco B_3 na Fig. 10.9(c), possui como variável básica de indução i, porque a única atribuição ao laço incrementa i de valor em 1. A família de i contém também t_4, t_6 e t_7. Por conseguinte, a tripla para t_4 é $(4, 0, 1)$. Semelhantemente, t_6 é da família básica de indução do laço, com a tripla $(6, 0, 1)$, e t_7 está na família de i, com a tripla $(4, 0, 1)$.

Podemos também procurar por variáveis de indução pelas outras extensões como j, exceto B_3 e B_4. Apenas j e t_2 são variáveis básicas de indução nesse laço maior. Os termos t_4 e t_8 são variáveis de indução com triplas $(4, 0, 1)$ e $(4, 0, 4)$, respectivamente.

O grafo de fluxo da Fig. 10.9(d) é obtido a partir daquele da Fig. 10.9(c) após aplicarmos o presente algoritmo. Descrevemos essa transformação abaixo.

Algoritmo 10.10 Redução de força aplicada às variáveis de indução.

Entrada. Um laço L com informações das definições incidentes e das famílias de variáveis de indução, computadas usando o Algoritmo 10.9.

Saída. Um laço revisado.

Método. Considerar cada variável básica de indução i de cada vez. Para cada variável de indução j na família de i com tripla (c, d), fazer o seguinte:

a) Criar uma nova variável s, exceto que se $j = i$ e $d = 0$, s e j terão triplas iguais, e não somente uma nova variável para cada uma.

b) Substituir a atribuição $j = e$ por $j = s$.

c) Imediatamente após cada atribuição $i = i + n$ em L, onde n é uma constante, anexar

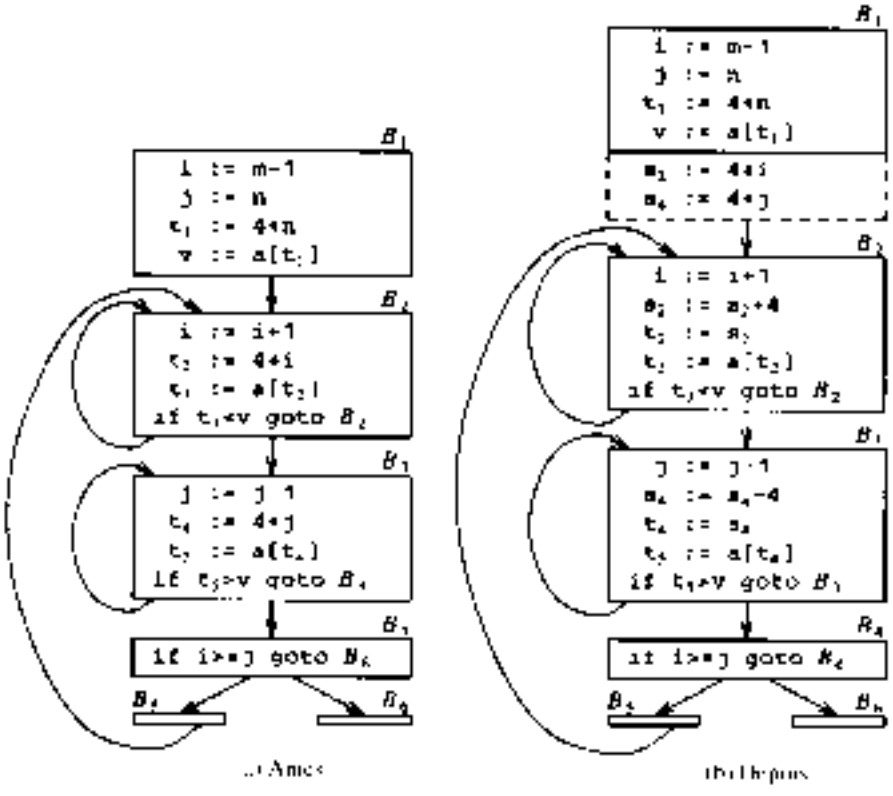
$$s = s + cn$$

onde a expressão cn é avaliada como uma constante ou como uma variável não-básica. Deixar a tripla de s como (c, d).

d) Resolver se s será inicializada antes de entrar no laço L. A inicialização pode ser necessária no final de um caminho. A inicialização consiste em

---

[1] Uma exceção será o caso de $k = j / b$, onde b é uma constante e j é uma variável básica de indução. Neste caso, k não pertence à família de j, porque $k = (1/b)j$ não é uma expressão linear de j se b for um inteiro.



(a) Antes (b) Depois

Fig. 10.39. Redução de capacidade.

[illegible] Note-se que [illegible] uma variável de indução da família de [illegible]

Exemplo 10.24. Suponhamos que consideramos os laços da Fig. 10.39(a), de dentro para fora. Como anteriormente, os laços mais internos contendo B_2 e B_3 é muito pobre. Falemos somente sobre o laço que envolve B_2. No Exemplo 10.23, observamos que a variável básica de indução no laço em torno de B_2 é i, e que a outra variável de indução é t_2, com tripla $(i, 4, 0)$. No passo 1 do Algoritmo 10.12 uma nova variável t_6 é constituída. No passo 2a a atribuição $t_2 := t_2 + 4$ é substituída por $t_6 := t_6 + 4$. O passo 4 insere a atribuição $t_6 := 4 * i$ após a atribuição $i := i + 1$. O valor de t_6 é obtido multiplicando-se o valor da atribuição $i := i + 1$ na tripla $(i, 4, 0)$ para t_2.

Como B_3 serve como um pré-cabeçalho para o laço, podemos colocar a inicialização de t_6 ao final do bloco B_3 que contém a definição de t_2. As instruções adicionadas são mostradas, na extensão pretendida, do bloco B_3.

Quando um laço mais externo é considerado, o código de fora se parece com o mostrado na Fig. 10.39(b). Existem quatro variáveis [illegible] que poderiam ser consideradas variáveis de indução. No entanto, o passo 3 do Algoritmo 10.12 eliminou as quatro recém-criadas que são [illegible] respectivamente, para facilitar a eliminação de [illegible] quando o processo [illegible]

Após a redução de capacidade encontramos que o único uso de alguma variável de indução está em testes, pode-se substituir um teste de uma tal variável por aquele de uma outra. Por exemplo, se i e t são variáveis de indução, tais que a cada vez sempre que i toma o valor de t é igual a t, então o teste $i = t$ é equivalente a $t = t_0$. Após essa substituição, pode ser possível eliminar i. Note-se, entretanto, que se i [illegible] podemos necessitar modificar o operador relacional igualmente, porque $i < t$ é equivalente a $t > t_0$. Seu algoritmo seguinte considera [illegible] o caso onde a constante multiplicativa k para t é positiva e a generalização para as constantes negativas como um exercício.

Algoritmo 10.11 Eliminação das variáveis de indução.

Entrada. Um laço L com informações sobre definições mediantes cópias [illegible] fazem [illegible] Algoritmo 10.7 e variáveis vivas.

Saída. O laço reescrito.

Método

1. Considerar cada variável básica de indução i cujas [illegible] jam partes apropriadas [illegible] de indução em sua família em [illegible] condicionais. Considerar algum [illegible] tal que i pode [illegible] velmente ainda ser [illegible] Se i e t [illegible] são ambas [illegible] simples possível teste [illegible] $i < t$ ou $i = t$. Se um [illegible] ficamos cada teste em que i aparece [illegible] modo a usar [illegible] Assumamos, no que se segue, que i [illegible] positivo. Um teste da forma i **op-relacional** x, onde x não seja uma variável de indução é substituído por

```
    r := x * k      /* omitir se k for 1 */
    r := r + c      /* omitir se c for 0 */
    op-relacional : t ... r
```

onde r é uma nova variável temporária. O caso x **op-relacional** i é tratado analogamente. Se existirem duas variáveis de indução i e j no teste i **op-relacional** j, [illegible] checamos cuidadosamente se ambas i e j podem ser substituídas. O caso mais simples é quando temos [illegible] com tripla (i, c_1, d_1) e j com tripla (j, c_2, d_2), $c_1 = c_2$ e $d_1 = d_2$. Então i **op-relacional** j é [illegible] substituído [illegible] **op-relacional** t. Em casos mais complexos, a substituição do teste pode não valer a pena porque poderemos necessitar introduzir dois passos multiplicativos e um aditivo, enquanto esses dois passos podem ser comunicados eliminando-se i e j.

Finalmente, remover todas as atribuições às variáveis de indução eliminadas [illegible] do laço L, na medida em que aquelas serão mais

2. Consideremos agora cada variável de indução i para as quais um enunciado $i := i + k$ foi gerado pelo Algoritmo 10.10. Em resumo, verificamos que podem não haver atribuições a i entre o enunciado considerado e o início do bloco básico. Nessa situação, é usada uma busca no bloco que determina simplificando essa verificação; em caso contrário, as informações a respeito das definições mediantes que algum cálculo do grau é necessitada para implementar a verificação. Substituímos todos os usos de i por usos de t e removemos o enunciado $i := i + k$.

Exemplo 10.25. Consideremos o grafo de fluxo da Fig. 10.3(b). O laço mais interno em torno de B_3 contém duas variáveis de indução j e t_4, mas nenhuma das duas pode ser eliminada porque t_4 não pode ser usada num teste fora do laço. Semelhantemente, o laço em torno de B_2 contém as variáveis de indução i e t_2, mas nenhuma das duas pode ser eliminada.

Vamos aplicar o Algoritmo 10.11 ao laço mais externo. Quando as novas variáveis s e t_5 forem criadas pelo Algoritmo 10.10, considerará os Exemplo 10.24 e 10.25 e colocará a tarefa de s e t_5 na A. Consideremos a tarefa de t_5. O único uso de t_5 no teste para terminação do laço no bloco B_3 é, desse modo, $s < t_5$. Uma candidata para eliminação no passo 1 do Algoritmo 10.11 é t_5. O teste no bloco B_3 envolve as duas variáveis de indução s e t_5. Felizmente, as tuplas de s e t_5, $(s, B_3, 4, 0)$ e $(t_5, B_3, 4, 0)$, respectivamente, têm os mesmos valores constantes em suas tuplas, pois que as tuplas são $(i, B_2, 4, 0)$ e $(t_4, B_3, 4, 0)$, respectivamente. Desse modo, podemos, por conseguinte, substituir s por t_5, permitindo que s seja eliminada.

O passo 2 do Algoritmo 10.11 aplica a propagação de cópias às variáveis recém-criadas, substituindo s e t_5 por t_4 e t_2, respectivamente.

Variáveis de Indução com Expressões Laço-Invariantes

Nos Algoritmos 10.8 e 10.10, podemos permitir expressões laço-invariantes em lugar de constantes. Um enunciado tripla $i := c$ em uma variável de indução i pode conter expressões laço-invariantes em vez de constantes. A avaliação dessas expressões deveria ser realizada fora do laço, no pré-cabeçalho. Sobretudo, uma vez que desejamos ter instruções inteiras, temos que ter cuidado para que a multiplicação por k não produza um operador não instrução. Precisamos estar preparados para gerar os enunciados do código intermediário para a avaliação de expressões. A substituição dos testes no Algoritmo 10.11 requer que a soma da constante multiplicativa e seja conhecida. Por esta razão, pode ser necessário restringir a aplicação dos casos em que c seja uma constante concreta.

10.8 LIDANDO COM PSEUDÔNIMOS

Se duas ou mais expressões denotam a mesma localização de memória, dizemos que as expressões são apelidos uma da outra, ou, precisamente, que têm a mesma localização. Nesta seção, temos considerado a análise de fluxo de dados na presença de apontadores e procedimentos, ambos introduzindo pseudônimos.

A presença de apontadores torna a análise de fluxo de dados mais complexa, pois que não sabemos o que pode ser apontado. A menos que façamos suposições sérias sobre onde um dado apontador pode apontar, é necessário assumir que uma atribuição indireta através de um apontador possa potencialmente modificar todo e qualquer variável. Precisamos assumir, também, que qualquer uso de dados endereçados por um apontador, como, por exemplo, $x := *p$, pode potencialmente usar qualquer variável. Essas suposições resultam em muitas variáveis vivas e definições incidentes e menos expressões disponíveis do que seria, de outro modo, possível. Felizmente, pesquisas sobre análise de fluxo de dados para indicar para o que um apontador poderia apontar permitiram-nos, por conseguinte, obter uma informação mais acurada a partir de nossas análises de fluxo de dados.

Quando uma atribuição envolve uma variável local, ou explicitamente uma chamada de procedimento, nós normalmente supomos que realizar uma suposição do pior caso — a de que tudo pode ser modificado — seja possível computar o conjunto de variáveis que o procedimento possa mudar. Como com todas as otimizações de código, podemos ainda conter-nos dentro das suposições conservativas: isto é, os conjuntos de variáveis cujos valores "possam ser" modificados quando devam incluir aproximadamente as variáveis que tenham efetivamente modificadas em todas as execuções do programa. Idealmente, como de praxe, simplesmente tentar chegar razoavelmente próximos aos verdadeiros conjuntos de variáveis modificadas e usadas, sem trabalhar excessivamente ou inadequadamente, ou cometendo erros que alterem o que o programa realiza.

Uma Linguagem Simples de Apontadores

Por uma questão de especificidade, vamos considerar uma linguagem simples. Ela possui tipos de dados elementares, por exemplo, inteiros e reais, reconhecidos uma pela outra em tempo de execução. Para cada tipo básico existe um tipo apontador para esse elemento básico e para arranjos unidimensionais desse elemento básico. Ficaremos sempre sabendo que um apontador está apontando para alguma localização de memória, mas não nos preocupamos com que elemento particular de dado está sendo apontado. Este agrupamento de todos os elementos apontados na memória em que cada um de apontadores estejam sendo considerados é possível. Tipicamente, os apontadores serão usados muito cuidadosamente para se percorrer todo um arranjo, de forma que uma análise de fluxo de dados mais detalhada, se pudermos conceber-la de todo, frequentemente poderia nos dizer que, de qualquer forma, a um ponto particular do programa, p poderia estar apontando para qualquer um dos elementos de A.

Precisamos também realizar certas suposições a respeito de que operações aritméticas sobre apontadores são semanticamente significativas. Primeiro, se um apontador p aponta para um elemento de dados primitivo, uma palavra, então qualquer operação aritmética sobre p produz um valor que pode ser um inteiro mas não um apontador. Se p aponta para um arranjo, então uma adição ou subtração de um inteiro deverá p, apontando para algum lugar no mesmo arranjo, enquanto que outras operações aritméticas sobre apontadores produzem valores que não são apontadores. Conquanto nem todas as linguagens proíbam a movimentação de um apontador de um arranjo para um outro através de uma operação de apontador, tal ação seria dependente da implementação particular assegurar que o arranjo A se situasse na memória. Desse ponto de vista que um compilador otimizante deve manter-se fiel à definição de linguagem ao decidir que otimizações realizar. Cada implementação de compilador, no entanto, precisa fazer um julgamento sobre que otimizações específicas o compilador deve ou não ser permitido realizar.

Os Efeitos das Atribuições de Apontadores

Sob essas suposições, as únicas variáveis que poderiam ser possivelmente usadas como apontadores são aquelas declaradas como apontadores e às quais um valor que seja o apontador adicionado ou subtraído de uma constante. Iremos nos referir a tais variáveis como apontadores. Nossas regras para determinar para o que um apontador p pode apontar são como se segue:

1. Se existir um enunciado de atribuição $q := r$ ou $q := r + c$ imediatamente após o ponto n, no bloco B, para q. Se r for um apontador, então após qualquer atribuição a r da forma $r := r + d$, onde d seja uma constante, p poderia apontar para o mesmo lugar que r apontava antes da atribuição. Se r for um arranjo, então q poderia apontar para o próximo elemento do arranjo.

Nota de rodapé — [illegible]

2. Se existir um comando de atribuição $x := y$ ou $x := y + z$, onde x e y (ou z) forem de tipos apontadores, então, usualmente, após a execução do comando x poderá apontar para qualquer coisa que y (ou z) possa apontar através de uma cadeia de zero ou mais acessos.

3. Se existir uma atribuição $x := \&y$, então, imediatamente após a execução desse comando, x poderá apontar para y e para qualquer coisa que y possa apontar através de uma cadeia de zero ou mais acessos.

4. Depois de qualquer outro tipo de atribuição $x := \ldots$, não há objeto que x possa apontar. Essa suposição pode parecer um tanto drástica, mas ela é razoável quando consideramos a semântica de linguagens com segurança de tipos.

5. Após qualquer chamada de procedimento, x pode apontar para qualquer objeto que pudesse apontar antes da chamada. Notemos que isso não pressupõe que um procedimento sempre possa apontar para um outro apontador. O relacionamento desta suposição com a forma de passar parâmetros para procedimentos é discutido, generalização para o caso de

Iniciemos definindo $entrada[B]$ para um bloco B, como sendo o único estado que fornece, para cada apontador x, o conjunto de variáveis para as quais x poderia estar apontando no início de B. Suponhamos que $entrada[B]$ é um conjunto de pares da forma (x, a), onde x é um apontador e a uma variável, significando que x poderia apontar para a.

Na prática, $entrada[B]$ poderia ser representado como uma lista para cada apontador, a lista para x que forneceria o conjunto de a's tais que (x, a) está em $entrada[B]$. Definimos $saída[B]$ similarmente para o final de B.

Especificamos a função de transferência para determinado bloco B através de B. Isto é, $entrada$ é uma função cujo argumento é um conjunto de pares (x, a), tais que x é um apontador e a uma variável, significando que x pode apontar para a. Presumimos, inicialmente, que $entrada[B]$ é o conjunto vazio. Presumimos, também, que $entrada$ é monotônica, isto é, se $S \subseteq T$, então $entrada(S) \subseteq entrada(T)$. Notemos que, se $entrada[B]$ é o resultado da aplicação seja $saída[B]$. Notemos também que podemos dizer como computar todos os $entrada$'s simultaneamente, $entrada$, será, por conseguinte, uma composição de $entrada$ para cada enunciado de bloco B. As regras para se computar $entrada$ são como se segue.

1. Se $x := \&y$ ou $x := y + z$, para qualquer x apontador, então

$$entrada(x) = \{(x, v) \mid (y, v) \text{ qualquer variável}\} \cup \{(x, v) \mid (z, v) \text{ qualquer variável}\}$$

2. Se for $x := *y$, para x um apontador e y um apontador, então

$$entrada(x) = \{(x, v) \mid (y, w) \text{ qualquer variável}\} \cup \{(x, v) \mid (y, w) \text{ qualquer variável}\}$$

Notemos que esta regra faz sentido mesmo se

3. Se $x := \&y$, então

$$entrada(x) = \{(x, v) \mid (y, v) \text{ qualquer variável}\} \cup \{(x, y)\}$$

4. Se x não for um apontador ou qualquer outra expressão, então

$$entrada(x) = \{(x, v) \mid (x, v) \text{ qualquer variável}\}$$

5. Se x não for uma atribuição a um apontador, então $entrada(x) = \emptyset$.

Podemos agora escrever as equações que relacionam $entrada$ e $saída$ como segue

$$saída[B] = transf_B(entrada[B])$$

$$entrada[B] = \bigcup_{P \text{ predecessor de } B} saída[P] \qquad (10.13)$$

onde, se B consiste dos enunciados $s_1, s_2, \ldots, s_n$, então

$$transf_B(S) = transf_{s_n}(transf_{s_{n-1}}(\ldots(transf_{s_1}(S))\ldots))$$

As equações (10.13) podem ser resolvidas essencialmente como as definições recursivas de Algoritmo 10.1. Não iremos, por conseguinte, entrar em detalhes do algoritmo, mas nos contentaremos com um exemplo.

Exemplo 10.26. Consideremos o grafo de fluxo da Fig. 9.4(b). Suponhamos que o único apontador seja p e que as únicas variáveis sejam a e b. Suponhamos, também, que $entrada[B_1]$ seja $\emptyset$ e que $entrada[B_2]$ possua o efeito de fazer com que qualquer variável apontada por p na entrada de B_2 seja apontada por p e b. Isto é, $entrada(B_2) = \{(p, a), (p, b)\}$.

Então, $entrada(B_1) = saída[B_1] = \emptyset$. Para todas as outras variáveis v e todos os pares com primeiro componente p, pelo par (p, v). O efeito de $p := \&a$ é de substituir pares com primeiro componente p por (p, a). Notemos que $p := \&a$ é efetivamente uma atribuição da forma $p := q$ para uma constante q. Podemos agora computar

$$saída[B_2] = entrada(B_2) \cup \{(p, a)\} = \{(p, a), (p, b)\}$$

Similarmente, $entrada[B_3] = \{(p, a)\}$ e $saída[B_3] = \{(p, a), (p, b)\}$.

Em seguida, encontramos que $entrada[B_4] = saída[B_2] \cup saída[B_3] = saída[B_2]$. Presumivelmente, $saída[B_4]$ foi inicializado em $\emptyset$ e, portanto, não foi modificado nessa passagem ainda. No entanto, $saída[B_4] = \{(p, a), (p, b)\}$, e $saída[B_4] = \{(p, a), (p, b)\}$. Todos os demais são iguais.

$$entrada[B_5] = \{(p, a), (p, b), (q, a), (q, b)\}$$

O efeito de $p := q$ em B_5 é de descartar a possibilidade de p não apontar para a ou b. Isto é

$$saída[B_5] = (entrada(B_5) - \{(p, a), (p, b)\}) \cup \{(p, a), (p, b)\}$$

Notamos, então, que sempre que B_5 for executado fazendo p apontar para a, a está semanticamente sendo atribuído (ou fornecido) indiretamente através de q em B_5. Por conseguinte, esse efeito é "preservado" mas efetivamente ilustra as dificuldades que podemos fazer a respeito dos apontadores.

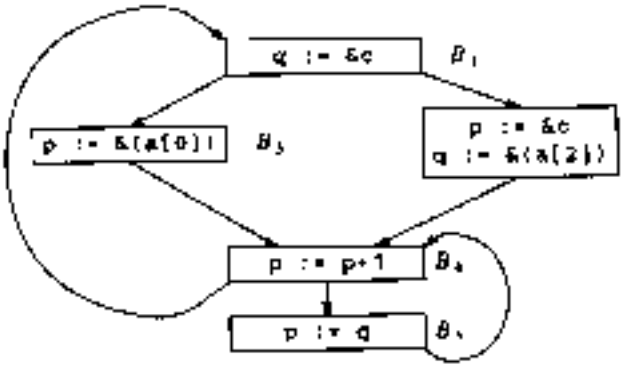


Fig. 10.40. Grafo de fluxo com duas variáveis apontadoras e dois apontadores.

Continuando, $entrada[B_3] = saída[B_2]$ e [illegible] de [illegible] e os da igualmente a [illegible]. Uma vez que [illegible] pode apontar somente para [illegible] ou [illegible] em $entrada[B_3]$,

$$saída[B_3] = \{ \ldots \}$$

Na passagem seguinte, encontramos que $entrada[B_2] = saída[B_1] \cup saída[B_3]$ e, por conseguinte, $saída[B_2] = \{ \ldots \}$. Este valor também aparece nos novos $entrada[B_3]$ e $entrada[B_2]$, mas esses novos valores não mudam $saída[B_3]$ ou $saída[B_2]$, nem é $entrada[B_2]$ modificada. Convergimos, dessa forma, para a resposta desejada.

Fazendo Uso das Informações sobre os Apontadores

Suponhamos que $entrada[B]$ seja o conjunto de variáveis apontadas por cada apontador no início do bloco B e que tenhamos uma referência ao apontador [illegible] dentro do bloco B. Começando por $entrada[B]$, aplicamos $trans_s$ para cada enunciado s do bloco B que preceda a referência a [illegible]. Este computo nos diz para o que [illegible] poderia apontar naquele enunciado particular onde aquela informação é importante.

Vamos supor agora que tenhamos determinado para o que cada apontador poderia apontar quando usado numa referência indireta, quer à esquerda ou à direita do sinal de atribuição. Como podemos usar esta informação para obter soluções mais acertadas para os problemas usuais de fluxo de dados? Em cada caso, precisamos considerar em que direção os erros são conservativos e precisamos utilizar as informações dos apontadores de forma em que somente erros conservativos sejam cometidos. [illegible] esta escolha é feita, vamos considerar dois exemplos: as definições incidentes e a análise das variáveis vivas.

Para calcular as definições incidentes, podemos usar o Algoritmo 10.2, mas necessitamos conhecer os valores para os conjuntos $mortas$ e $geradas$ de um bloco. O último [illegible] computado como de hábito para os enunciados que não sejam atribuições indiretas através de apontadores. Uma atribuição indireta [illegible] é considerada gerar uma definição para cada variável [illegible] para a qual [illegible] possa apontar. A suposição é conservativa, porque, como discutido na Seção 10.5, é geralmente conservativo assumir que as definições atingem um ponto ainda que, na realidade, não o façam.

Ao computarmos $mortas$, assumiremos que [illegible] mata definições de [illegible] somente se [illegible] não for um apontador e for a única variável para a qual [illegible] poderia estar apontando. Se [illegible] puder apontar para duas ou mais variáveis, então assumiremos que as definições de qualquer delas sejam mortas. De novo, estamos sendo conservativos porque permitimos que as definições de [illegible] [illegible] [illegible] e atinjam aquilo que quer que possam, a menos que possamos provar que [illegible] tenha redefinido [illegible]. Em outras palavras, quando temos dúvida, assumimos que uma definição incida.

Para as variáveis vivas, podemos usar o algoritmo 10.4, mas precisamos reconsiderar como $definições$ e $usos$ devem ser definidos para os enunciados da forma [illegible] e [illegible]. O enunciado [illegible] usa somente [illegible]. Dizemos que o mesmo define [illegible] somente se [illegible] for a única variável para a qual [illegible] possa apontar. Esta suposição permite que os usos de [illegible] passem, a menos que sejam seguramente bloqueados pela atribuição [illegible]. Por conseguinte, jamais poderemos afirmar que a variável [illegible] esteja morta em um ponto quando está de fato viva. O enunciado [illegible] certamente representa uma definição de [illegible]. Representa igualmente um uso de [illegible] e um uso de qualquer variável para a qual [illegible] poderia apontar. Maximizando os possíveis usos, de novo maximizamos nossas estimativas das variáveis vivas. Maximizando as variáveis vivas estamos normalmente sendo conservativos. Por exemplo, poderíamos gerar código para armazenar uma variável morta, mas jamais iremos deixar de armazenar para uma que estivesse viva.

Análise Interprocedimental do Fluxo de Dados[*]

Até agora, falamos de "programas" que eram procedimentos singelos e, consequentemente, grafos de fluxo simples. Iremos agora ver como capturar informações a partir de muitos procedimentos interagindo. A idéia é a seguinte: determinar como cada procedimento influencia as informações a respeito de $geradas$, $mortas$, $usos$ ou $definições$ dos demais procedimentos, e então computar nossas informações de fluxo de dados para cada procedimento como antes.

Durante a análise de fluxo de dados, teremos que lidar com os pseudônimos estabelecidos pelos parâmetros nas chamadas de procedimento. Como não é possível para duas variáveis globais denotarem o mesmo endereço de memória, pelo menos um deles de um par de pseudônimos terá que ser um parâmetro formal. Como os parâmetros formais poderão ser transmitidos para procedimentos, é possível que dois parâmetros formais sejam pseudônimos.

Exemplo 10.27. Suponhamos ter um procedimento [illegible] com dois parâmetros formais [illegible] e [illegible], transmitidos por referência. Na Fig. 10.[illegible] vemos uma situação na qual [illegible] e [illegible] é computado em B_1 e B_2. Suponhamos que os únicos percursos a partir de bloco B_1 para B_2 passem através de B_3 e que não existam atribuições a [illegible] ao longo de qualquer desses percursos. Está [illegible] disponível em B_2? A resposta depende de [illegible] e [illegible] poderem denotar o mesmo endereço de memória. Por exemplo, poderia haver uma chamada [illegible] ou talvez uma chamada de [illegible], [illegible] e [illegible] fossem parâmetros formais de outro procedimento [illegible] e uma chamada [illegible] ser possível.

Similarmente, é possível para [illegible] e [illegible] serem pseudônimos se [illegible] for um parâmetro formal, digamos de [illegible], e [illegible] uma variável com um escopo acessível a algum procedimento [illegible] que chama [illegible] digamos, através de uma chamada [illegible]. Situações ainda mais complicadas podem tornar [illegible] pseudônimos e iremos desenvolver em breve algumas regras gerais para determinar todos esses pares de pseudônimos. □

Iremos estabelecer algumas situações em que seria conservativo não considerar os nomes de variáveis como pseudônimos. Por exemplo, nas definições incidentes, se desejamos estabelecer que uma definição de [illegible] é morta por uma definição de [illegible], melhor teremos então assegurar que definitivamente [illegible] e [illegible] serão pseudônimos sempre que a definição de [illegible] for executada. Em outras situações, seria conservativo considerar os nomes de variáveis como pseudônimos sempre que houver dúvida. O Exemplo 10.27 é um desses casos. Se a expressão disponível [illegible] não deve ser morta por uma definição de [illegible], melhor faremos em assegurar que nem [illegible] nem [illegible] possam ser pseudônimos de [illegible].

Um Modelo de Código com Chamadas de Procedimentos

Para ilustrar como pseudônimos [illegible] lidar com a pseudonimia, vamos considerar uma linguagem que permite procedimentos recursivos, alguns dos quais possam se referir tanto a variáveis locais quanto globais. Os dados disponíveis a um procedimento consistem somente nas variáveis globais e nas suas próprias variáveis locais, isto é, não há estrutura de blocos na linguagem. Os parâmetros são transmitidos por referência. Requeremos que todos os procedimentos tenham um grafo de fluxo com uma única $entrada$ e um único nó de $retorno$ que faz o controle retornar para o procedimento chamador. Supomos, por conveniência, que cada nó repouse em algum percurso de entrada para o retorno.

Suponhamos, agora, que estejamos num procedimento [illegible] e cheguemos a uma chamada de procedimento [illegible]. Se estivermos interessados em computar definições incidentes, expressões disponíveis ou qualquer outras análises de fluxo de dados, precisamos saber se [illegible]

[*] [illegible] interprocedimental [illegible] [illegible] Referências [illegible] [illegible] [illegible]

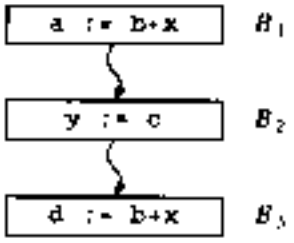


Fig. 9.41 [illegible] problemas [illegible].

pode mudar o valor de alguma variável. Notemos que dizemos "pode mudar" em lugar de "muda". Como em todos os problemas de fluxo de dados, é impossível saber com certeza se o valor de uma variável é modificado ou não. Podemos somente encontrar um conjunto que inclua todas as variáveis cujos valores podem realmente ser alterados e talvez algumas que não. Sendo cuidadosos, poderemos produzir a última classe de variáveis, obtendo aproximação da realidade o mais exata possível dentro do aspecto conservativo.

As únicas variáveis cujos valores poderão ser alterados pela chamada [illegible] serão as globais e as variáveis [illegible] que poderão ser locais a [illegible]. As definições das variáveis locais a [illegible] não produzirão consequências após o retorno das chamadas. Ainda que [illegible] contenham cópias das variáveis locais a [illegible], que poderão ser alteradas, essas cópias desaparecerão após o retorno. É fácil se determinar que variáveis globais são explicitamente definidas por [illegible] simplesmente vendo quais têm definições em [illegible] ou são definidas em uma chamada de procedimento feita por [illegible]. Adicionalmente, [illegible] que podem ser globais, mudar ou ser [illegible] por uma definição de seu parâmetro formal segundo parâmetros formais respectivamente correspondentes, [illegible] parâmetros formais foram transmitidos por referência para outro procedimento que as definiu. No entanto, nem todos os parâmetros estão modificados por uma chamada para [illegible] precisam ser definidos explicitamente por [illegible] por um dos procedimentos que o mesmo chame, pois as variáveis poderão ser pseudônimos.

Cômputo dos Pseudônimos*

Antes de podermos responder à questão sobre quais variáveis poderão mudar dentro de um procedimento, precisamos desenvolver um algoritmo para encontrarmos os pseudônimos. O enfoque que iremos usar aqui é simples. Computaremos a relação = sobre as variáveis que formalizam a noção "é um pseudônimo de". Agindo desse maneira, não distinguiremos as existências de uma variável em diferentes ativações do mesmo procedimento, apesar de diferenciarmos as variáveis locais a procedimentos diferentes. Então, nenhuma [illegible] identificado.

Para tornar as coisas mais simples, não tentaremos diferenciar os conjuntos de pseudônimos em diferentes pontos do programa. Ao invés, se duas variáveis puderem, de alguma forma, ser pseudônimas, assumimos que sempre o poderão ser. Finalmente, faremos a suposição adicional de que [illegible] transitiva, de forma que as variáveis serão agrupadas em classes de equivalência e duas variáveis poderão ser pseudônimas somente se estiverem na mesma classe.

Algoritmo 10.12 Cômputo simples de pseudônimos.

Entrada. Uma [illegible] de procedimentos e variáveis globais.

Saída. Uma relação de equivalência com a propriedade de que sempre que existir uma posição na qual a variável [illegible] pseudônimos [illegible], [illegible] isso não precisa ser verdadeira.

Método

1. Reconhecer as variáveis [illegible] de forma, que não [illegible] procedimentos que usem o mesmo parâmetro formal em diferentes chamadas de variável local e que não [illegible] variáveis globais, parâmetros formais ou variáveis globais compartilhando o mesmo declarador.

2. Se existir um procedimento [illegible], e uma invocação [illegible] desse procedimento fazer [illegible], para todo [illegible], cada parâmetro formal pode ser um pseudônimo de cada um de seus parâmetros atuais correspondentes.

3. Fazer o fechamento reflexivo e transitivo das correspondências entre os atuais e formais, adicionando

   [illegible] sempre que [illegible];
   [illegible], sempre que [illegible] para algum [illegible].

Exemplo 10.28. Consideremos os esqueletos dos três procedimentos apresentados na Fig. 10.42, onde é assumido que os parâmetros sejam transmitidos por referência. Existem duas variáveis globais, [illegible] duas variáveis locais. [illegible] procedimento [illegible] para [illegible] procedimento [illegible]. O procedimento [illegible] possui os parâmetros formais [illegible] e o procedimento [illegible] os formais [illegible]. [illegible] não possui parâmetros formais. Por conseguinte, nenhuma renomeação de variáveis é necessária. Computaremos primeiro a propagação devida às correspondências entre os parâmetros formais e atuais.

A chamada de [illegible] por [illegible] faz [illegible]. A primeira chamada de [illegible] por [illegible] faz [illegible]. A segunda, [illegible].

A chamada de [illegible] por [illegible] faz [illegible]. Os seguintes, neste exemplo, são fechamento transitivo dos relacionamentos entre os pseudônimos, representado por [illegible]. Convém notar que todos os pares de variáveis são possíveis pseudônimos nos exemplos. ❑

O cômputo que pelo algoritmo 10.12 não resulta frequentemente um grau de [illegible] como encontrado no Exemplo 10.28. Inadvertidamente, nós determinamos especificamente, [illegible] duas variáveis de tipos diferentes sejam variáveis uma da outra. Sobretudo, o programador pretende [illegible] que [illegible] certos - para essas variáveis. Por exemplo, se o primeiro parâmetro formal de um procedimento [illegible] representa uma velocidade pode ser

```
global g, h;
    procedure main( );
        local i;
        g := ... ;
        one(h, i)
    end
    procedure one(w, x);
        x := ... ;
        two(w, w);
        two(g, x)
    end;
    procedure two(y, z);
        local k;
        h := ... ;
        one(k, y)
    end
```

Fig. 10.42. Procedimentos de exemplo.

Optimizing compilers for modern architectures: a dependence-based approach [illegible] Allen and Ken Kennedy, Morgan Kaufmann Publishers, 2002, ISBN 1-55860-286-0.

```
11  para cada procedimento p faça modificados[p] ← ∅; alterados ← ∅
12  enquanto o conjunto alterados for diferente de modificados[p] faça
13      para cada procedimento q chamado por p faça início
14          adicionar quaisquer variáveis globais em modificados[q] a modificados[p];
15          para cada parâmetro formal x de q faça
16              se x estiver em modificados[q] então
17                  para cada chamada de q por p faça
18                      se y for o parâmetro real da chamada,
19                          e for global ou parâmetro formal de p
20                              então adicionar y a modificados[p];
    fim
```

Fig. 10.43 Algoritmo para computar modificação.

esperado que o primeiro argumento em qualquer chamada a `P` seja ajustado pelo programador como uma velocidade. Por conseguinte, esperamos fortemente que o primeiro parâmetro formal de `P` não seja modificado.

Análise de Fluxo de Dados na Presença de Chamadas de Procedimentos

Vamos considerar, por exemplo, como as expressões disponíveis podem ser calculadas na presença de chamadas de procedimentos, onde os parâmetros são transmitidos por referência. Como na Seção 10.6, precisamos determinar quando uma variável pode ser definida indiretamente por conseguinte, como as expressões precisam ser determinadas quando as expressões são geradas ou eliminadas.

Podemos determinar o efeito indireto de um procedimento `p` computando um conjunto *modificados*(`p`). Esse valor é o conjunto de variáveis globais e parâmetros formais de `p` que podem ser modificados depois que `p` é executado. Neste ponto, não contamos uma variável como modificada se um membro de sua classe de equivalência de possíveis sinônimos é modificado.

Seja *definidos*(`p`) o conjunto de parâmetros formais e variáveis globais que recebem definições explícitas dentro de `p`. Isso inclui, naturalmente, aqueles que são definidos dentro de procedimentos chamados por `p`. Podemos escrever as equações para *modificados* como segue. Estamos somente interessados nos nomes de variáveis globais e nos parâmetros formais de `p` que sejam usados como parâmetros reais em chamadas feitas por `p` aos parâmetros formais correspondentes dos procedimentos chamados. Podemos escrever

$$modificados(p) = definidos(p) \cup \bigcup_{q \text{ chamado por } p} modificados(q) - locais(q) \qquad (10.14)$$

onde

1. $locais(q)$ é uma variável global ou parâmetro formal de q tal que, para algum procedimento r referenciado em q, uma chamada $r(x)$ existe para um parâmetro real x e o parâmetro formal de r existe em $modificados(r)$.

2. q é chamado por p em algum programa que chama p.

Não devemos ser surpresos que a Equação (10.14) possa ser resolvida para um conjunto de procedimentos através de uma técnica iterativa. A precisão da solução não será diminuída, pois vamos somente da menor. Podemos começar a solução começando com uma aproximação bem pequena e iterando. A aproximação sem pesquisa sobre a qual começar é *modificados*(`p`) = *definidos*(`p`). Os detalhes da iteração são deixados como um exercício para o leitor.

É valioso considerar a ordem na qual os procedimentos devem ser visitados durante a iteração. Por exemplo, se os procedimentos não são mutuamente recursivos, podemos evitar por inteiro aqueles que não contribuam com definições para seus chamadores. Para esses procedimentos, *modificados* = *definidos*. Em seguida, podemos computar *modificados* para aqueles procedimentos que chamam somente procedimentos que não chamam mais ninguém. Podemos aplicar (10.14) para esse grupo seguinte de procedimentos diretamente, já que *modificados*(`q`) será conhecido para qualquer `q` em (10.14).

Esta idéia pode ser tornada mais precisa como se segue. Descobrimos um *protocolo chamador*, uma ordem de procedimentos tal que todas as chamadas `p(x)` para `q` ocorrem antes de `q` chamar `r`. Uma coleção de procedimentos que não são parte de nenhum ciclo de chamadas tem um protocolo chamador. Nesse caso, visitamos os nós somente uma vez.

Fornecemos agora um algoritmo para computar *modificados*.

Algoritmo 10.13 Análise interprocedimental das variáveis modificadas.

Entrada. Uma coleção de procedimentos `p(x, y, ...)`. Se o código de chamada for conhecido, assumimos que `p` chame somente se `p` é visível. Caso contrário, não fazemos suposição a respeito de que procedimento `p` chama ou quê.

Saída. Para cada procedimento `p`, produzimos *modificados*(`p`), o conjunto de variáveis globais e parâmetros formais de `p` que podem ser modificados explicitamente por `p` ou seus procedimentos.

Método.

1. Computar *definidos*(`p`) para procedimento `p` por inspeção.

2. Executar o programa da Fig. 10.43 para computar *modificados*. ☐

Exemplo 10.29. Vamos considerar a Fig. 10.42 de novo. Por inspeção, *definidos*(`f`) = {`f`, `x`}, *definidos*(`g`) = {`g`, `y`}, e *definidos*(`h`) = {`h`, `z`}. Esses são os valores iniciais de *modificados*. O grafo de chamadas dos procedimentos é mostrado na Fig. 10.44. Se nos considerarmos `f` primeiro, encontramos `g` e `h` como chamados por `f`. Como `f` não chama ninguém, o passo 1 não altera *modificados*(`f`). Na linha (6), precisamos considerar somente o segundo parâmetro formal de `g`, pois `x` é o primeiro parâmetro formal correspondente a `x`, o único modificado. Consequentemente, fazemos *modificados*(`f`) igual a {`f`, `x`, `y`} na linha (10).

Consideramos agora `g`. Na linha (5), `g` pode somente ser chamado por `f` e `h`.

Assim, de maneira geral, o código de um procedimento pode ser otimizado se soubermos que as únicas chamadas a esse procedimento são aquelas em que o segundo parâmetro real é o mesmo que a variável global `y`, e não há chamadas indiretas a esse procedimento.

Fig. 10.44 Grafo de chamadas

A linha 5 é global em *modificados*[ ] e assim fazemos *modificados* = { }. As duas chamadas de proc no parâmetro formal de ... estão em *modificados*[ ], de modo que descobrimos possíveis aliasings adicionais e *modificados* = { }. Pela Tabela 10.1, sendo esses os dois primeiros parâmetros atuais nas chamadas de procedimento ... Por conseguinte *modificados* = { }.

Consideremos, agora, ... O procedimento ... modifica ambos os seus parâmetros formais e, por conseguinte, ambos ... estão modificados na chamada de ... por parte ... No entanto, é uma variável local e não precisa ser considerada. Consequentemente, fazemos *modificados* = { }. Finalmente, repetimos o laço enquanto da linha 2. As seguintes alterações ... são propagadas quando modificamos a variável global ... Por conseguinte, a chamada para ... faz com que ... seja modificada através de *modificados*[ ] = { }. Não ocorrem modificações adicionais na iteração.

Uso das Informações do Conjunto Modificados

A maneira como o conjunto *modificados* pode ser usado, considerando o cômputo global das subexpressões comuns. Vamos supor que estamos computando as expressões disponíveis para o procedimento ... e desejamos computar *anavez*[*B*] para um bloco *B*. Uma definição da variável ... precisa ser encontrada, a menos que ... seja um parâmetro formal de ... e não esteja em *modificados*. Uma vez que ... será um pseudônimo de memória que ... e um pseudônimo de ... mesmo que ... não seja em *modificados*. Por conseguinte, as informações computadas pelos Algoritmos 10.1 e 10.15 podem ser usadas para computar uma aproximação segura do conjunto de expressões inutis.

Para computar as expressões disponíveis nos programas com chamadas de procedimentos, precisamos também ter uma forma conservativa de estimar o conjunto de expressões geradas por uma chamada de procedimento. Para sermos conservativos, poderemos assumir que ... seja gerada por uma chamada para ... se e somente se ... em cada percurso da entrada de ... até uma saída, encontramos ... sem nenhuma redefinição subsequente de ... Quando procuramos por ocorrências de ..., não podemos escolher ... como uma tal ocorrência a menos que estejamos seguros de que cada chamada de ... levará a uma ocorrência pseudônima de ... ou de ....

Estamos essencialmente repetindo a discussão anterior sobre quando uma expressão não está disponível ao redor de uma chamada. No presente caso, precisamos assumir que ... seja morta ao redor de qualquer ... que pudesse possivelmente ser um pseudônimo de ... ou ... Por conseguinte, a hipótese para computar as expressões disponíveis para todos os nós de todos os procedimentos é assumir que uma chamada não gera nada, ou seja, *gerado*[*B*] para todos os blocos *B* é computado como ausência. A medida que vai se especializando as expressões sejam geradas por um procedimento tipicamente vai tornar o fluxo suficiente para a maior parte dos propósitos.

Uma alternativa um pouco mais complicada é computar para cada procedimento as expressões disponíveis e de ocupado ao nível [ ] para cada procedimento, novamente. Podemos então utilizar *gerado*[ ] como o conjunto de expressões disponíveis ao fim do nível de ... de acordo com o modelo acima. Isso ... melhora polinomialmente o pior caso para as expressões geradas. ... representa somente as ocorrências ... que não são precedidas por ... ou ... que possam ser pseudônimos de ... ou de ....

Computemos agora as expressões disponíveis para Exemplo 10.30 de acordo com os elementos de nível ... Nenhuma chamada ... gera quaisquer expressões em *gerado*. [ ] para ... é substituído pelos parâmetros formais correspondentes de ... e *gerado* permanece como antes. Um novo valor de *gerado*[ ] para cada procedimento ... pode ser encontrado olhando-se para quais expressões estão disponíveis ao final do retorno de .... Esta iteração pode ser repetida até que nenhum dos *gerados* se altere. Todas as expressões disponíveis aqui são ...

10.9 ANÁLISE DE FLUXO DE DADOS DE GRAFOS DE FLUXO ESTRUTURADOS

Programas escritos em linguagens de alto nível possuem grafos de fluxo de um tipo especial, pois eles são projetados para facilitar a tradução dirigida pela sintaxe. Vários estudos de populares classes de programas têm revelado que virtualmente todos os programas escritos por pessoas têm grafos de fluxo que são redutíveis. Esta observação é relevante para fins de otimização, porque podemos encontrar algoritmos simples que trabalham rapidamente sobre grafos de fluxo redutíveis. Nesta seção, discutiremos uma variedade de conceitos de grafos de fluxo, incluindo vários tipos de intervalos, que são primariamente relevantes para os grafos de fluxo estruturados. Em seguida, veremos como as técnicas de tradução dirigida pela sintaxe desenvolvidas na Seção 10.5 podem ser generalizadas para incluir uma análise mais geral onde a sintaxe não precisa ser mais diretamente a estrutura do grafo de fluxo.

Busca em Profundidade

Existe um ordenamento útil dos nós num grafo de fluxo, muitas vezes chamado *ordenamento em profundidade*, que é uma generalização da travessia em caminhamento em profundidade de uma árvore, introduzida na Seção 2.3.3. Um ordenamento em profundidade pode ser usado para detectar laços em qualquer grafo de fluxo, tendo muitas propriedades relacionadas com algoritmos de busca de dados, tais como os discutidos na Seção 10.6. O ordenamento em profundidade é criado por expandir-se por toda a parte e buscando-se todo o grafo, lembrando-se quais nós já foram visitados, de modo que aqueles que não foram ainda visitados possam ser visitados mais tarde. Antes de darmos a seguir o algoritmo, vamos considerar um exemplo.

Exemplo 10.30 Uma possível busca em profundidade do grafo de fluxo da Fig. 10.45 é ilustrada na Fig. 10.46. Os laços contínuos formam uma árvore, e todos os demais nós são descendentes daquela árvore. A busca em profundidade do grafo de fluxo corresponde a um caminhamento pré-ordem da árvore. Começamos pelo nó 1 e vamos para 2 e então para 3, e 4. Do nó 4 podemos ir para 1 ou 5, mas como já visitamos 1, vamos somente para 5. Do nó 5 podemos ir para 2 ou 6, mas como já visitamos 2, vamos somente para 6. Do nó 6 podemos ir para 3 ou 7, mas como já visitamos 3, vamos somente para 7. Finalmente, do nó 7 podemos ir para 4 ou 8, mas como já visitamos 4, vamos somente para 8. Do nó 8 podemos ir para 5 ou 9, mas como já visitamos 5, vamos somente para 9. Voltamos para 8 mas como já visitamos todos os seus sucessores, retrocedemos para 7, e daí para 6, 5, 4, 3, 2 e 1. Retiramos o nó 5 para 4, então de volta para 3 e 2. De 2 vamos para 2 retrocedemos para 1 e de volta para 1 e terminamos a busca em profundidade. Notemos que ainda não explicamos como a árvore é selecionada a partir do grafo de fluxo. ...

A *ordenação em profundidade* dos nós é o reverso da ordem na qual os nós são ...; por último os nós no caminhamento pós-ordem ...

Exemplo 10.31 No Exemplo 10.30, a sequência completa com nós retirados à medida que as pilhas ficam vazias é

1, 5, 4, 6, 7, 8, 10, 8, 9, 8, 7, 6, 3, 5, 2, 3, 1, 2, 1

Existe um pequeno problema com a definição de ordenação em profundidade, pois ela não é única. Isso ocorre porque a ordem em que os nós são visitados e a ordem em que os sucessores de um nó são visitados ...

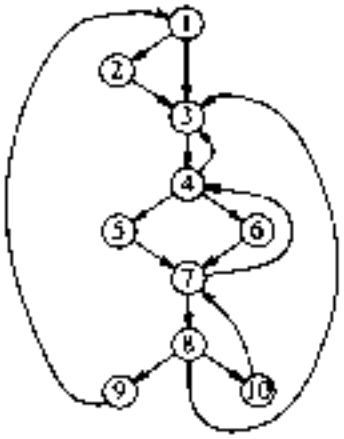


Fig. 10.45. Grafo de fluxo.

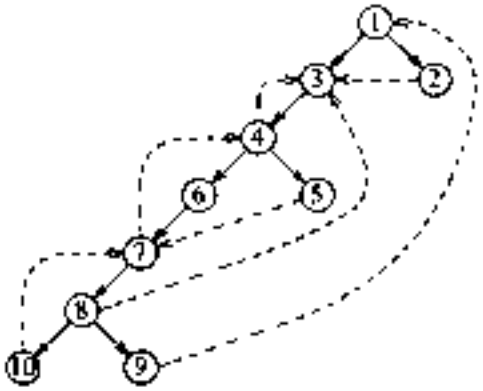


Fig. 10.46. Aprofundidade da busca em profundidade.

Nesta lista, anotamos a última ocorrência de cada número para obtermos

1, 2, 4, 6, 7, 8, 10, 8, 9, 8, 7, 6, 4, 5, 4, 3, 1, 2, 1

O ordenamento em profundidade é a sequência de números marcados na ordem reversa. Aqui, esta sequência é 10, 9, 8, 7, 6, 5, 4, 3, 2, 1. Isto é, inicialmente os nós foram numerados numa ordem em profundidade.

Daremos um algoritmo que computa o ordenamento em profundidade de um grafo de fluxo pela construção recursiva de uma árvore enraizada no nó inicial, tentando construir percursos na árvore na medida do possível. Uma tal árvore é chamada de *árvore de alcance em profundidade* (*depth-first spanning tree*). Daremos o algoritmo usado para construir a Fig. 10.46 a partir da Fig. 10.45.

Algoritmo 10.14 Árvore de alcance em profundidade e ordenamento em profundidade.

Entrada. Um grafo de fluxo G.

Saída. Uma árvore T de G e um ordenamento dos nós de G.

Método. Usaremos primeiramente o procedimento *pesquisar* da Fig. 10.47, o algoritmo inicializa todos os nós de G como "não visitados" e, em seguida, chama *pesquisar*(n_0), onde n_0 é o nó inicial. As chamadas *pesquisar*(n) marcarão primeiro n como "visitado", a fim de evitar que n seja visitado novamente duas vezes. Usaremos i para contar a partir do número dos nós de G até 1, atribuindo número de profundidade $np[n]$ aos nós n, à medida que seguimos em frente. O conjunto de lados T forma a árvore de alcance em profundidade para G, sendo os lados denominados de *lados da árvore*.

Exemplo 10.32. Consideremos a Fig. 10.47. Fazemos i igual a 10 e chamamos *pesquisar*(1). À linha 12 de *pesquisar* processamos consecutivamente cada sucessor do nó 1. Suponhamos que consideremos $s = 3$, primeiro. Adicionamos, então, o lado $1 \rightarrow 3$ a T e chamamos *pesquisar*(3). Em *pesquisar*(3), adicionamos o lado $3 \rightarrow 4$ a T e chamamos *pesquisar*(4).

Suponhamos que em *pesquisar*(4), tentamos escolher $s = 6$, primeiro. Adicionamos, então, o lado $4 \rightarrow 6$ a T e chamamos *pesquisar*(6). Isso, por sua vez, nos faz adicionar $6 \rightarrow 7$ a T e chamar *pesquisar*(7). O nó 7 possui dois sucessores, 4 e 8. Mas 4 já está marcado como "visitado" por *pesquisar*(4) e, dessa forma, não fazemos nada quando $s = 4$. Quando $s = 8$, adicionamos o lado $7 \rightarrow 8$ a T e chamamos *pesquisar*(8). Suponhamos que escolhemos, então, $s = 10$. Adicionamos o lado $8 \rightarrow 10$ e chamamos *pesquisar*(10).

Agora, 10 possui um sucessor, 7, mas 7 já está marcado como "visitado" e, consequentemente, em *pesquisar*(10), saímos através do passo (6) da Fig. 10.47, fazendo $np[10] = 10$ e $i = 9$. Isto completa a chamada para *pesquisar*(10) e, por conseguinte, retornamos a *pesquisar*(8). Fazemos, agora, $s = 9$, em *pesquisar*(8), adicionamos o lado $8 \rightarrow 9$ a T e chamamos *pesquisar*(9). O único sucessor de 9, a nós 1, já está como "visitado". Fazemos, então, $np[9] = 9$ e $i = 8$. Retornamos, então, para *pesquisar*(8). O último sucessor de 8, o nó 3, é "visitado", e, portanto, não fazemos nada para $s = 3$. Assim, agora, tendo considerado todos os sucessores de 8, fazemos, então, $np[8] = 8$ e $i = 7$, e retornamos a *pesquisar*(7).

Todos os sucessores de 7 foram considerados e, por conseguinte, fazemos $np[7] = 7$ e $i = 6$, retornando a *pesquisar*(6). Similarmente os sucessores de 6 foram considerados, fazemos, então, $np[6] = 6$ e $i = 5$, e retornamos a *pesquisar*(4). O sucessor 3 de 4 já foi "visitado", mas 5 ainda não foi encontrado, então $4 \rightarrow 5$ a árvore e chamamos *pesquisar*(5), que não resulta em chamadas posteriores, na medida em que o sucessor 7 de 5 já foi "visitado". Pre consequinte, $np[5] = 5$, e i fica igual a 4 e retornamos a *pesquisar*(4). Tendo completado a consideração dos sucessores de 4, fazemos, então, $np[4] = 4$ e $i = 3$, e retornamos a *pesquisar*(3). Em seguida, fazemos $np[3] = 3$ e $i = 2$, retornando a *pesquisar*(1).

```
      procedimento pesquisar(n);
      início
(1)       marcar n como "visitado";
(2)       para cada sucessor s de n faça
(3)           se s é "não visitado" então início
                  adicionar lado n → s a T;
                  pesquisar(s)
              fim;
(6)       np[n] = i;
(7)       i = i - 1
      fim;
          { segue o programa principal }
(8)   T = vazio; { conjunto de lados }
(9)   para cada nó n de G faça marcar n "não visitado";
(10)  i = número de nós de G;
(11)  pesquisar(n0)
```

Fig. 10.47. Algoritmo de pesquisa em profundidade.

Depois de duas iterações, chamadas *passos*, o caminho [2, 4, 5, 6] terá sido encontrado. Ele estabelecerá $p[2] = 2$, $p[4] = 4$, $p[5] = 4$ e $p[6] = 5$. Observe que $p[3] = 1$ e $p[1] = 0$. Note que essas condições não são suficientes para garantir que $p[v] = v$ implica que v é um ancestral de w na árvore de busca em profundidade do grafo da Fig. 10.15. □

Lados numa Representação em Profundidade de um Grafo de Fluxo

Quando consideramos uma dada árvore de busca em profundidade de um grafo de fluxo, os lados do grafo de fluxo caem em três categorias:

1. Existem lados que são de um nó para um de seus ancestrais na árvore, incluindo o próprio nó. Esses lados serão chamados *lados de retorno*. Por exemplo, $3 \to 2$, $4 \to 4$ e $5 \to 1$ são lados de retorno na Fig. 10.16. É uma fato interessante e útil que, em qualquer grafo de fluxo, todos os lados retrocedentes são lados de retorno, independentemente da ordem na qual os sucessores sejam visitados no passo 2 da Fig. 10.17. Para mostrar que, em qualquer grafo de fluxo, qualquer lado retrocedente é de retorno, apesar de ser o grafo fortemente conexo, existem alguns lados retrocedentes que não são lados de retorno.

2. Existem lados, chamados de *avanço progressivos*, que vão de um nó para um de seus descendentes próprios na árvore. Todos os lados na árvore são lados progressivos. Não existem outros lados progressivos na Fig. 10.16, mas, por exemplo, se $a = 4$ fosse um lado, estaria nesta categoria.

3. Existem lados $v \to w$ tais que nem v é ancestral próprio de w nem w é ancestral próprio de v na árvore. Os lados $2 \to 3$ e $5 \to 2$ são os únicos da Fig. 10.16 que se encaixam nesta categoria. Chamamos tais lados de *lados cruzados*. Uma importante propriedade dos lados cruzados é que se desenharmos a árvore de busca em profundidade de forma que os filhos de um nó sejam colocados da esquerda para a direita na ordem em que são adicionados à árvore, todos os lados cruzados se deslocam da direita para a esquerda.

Devemos ser cuidadosos com $v = w$ ou um lado retrocedente se encaixa em mais de uma categoria? Exploramos [1] e [2]. Para ver por que nenhum dos outros pode ocorrer, observe que v é um descendente de w ou vice-versa, pois $p[v]$ é um dos ancestrais de v para os quais $p[v] \geq w[v]$. Respectivamente, se $w[v] < w[w]$, pois w é um dos ancestrais de v para os quais $p[v] \geq w[v]$. Vale ressaltar que precisamos corrigir $p[v]$ antes de passarmos ao lado $v \to w$, pois, caso contrário, o fato de v ser um sucessor de w em árvore não nos permitiria deduzir que w é um ancestral de v na árvore.

Profundidade de um Grafo de Fluxo

Existe um importante parâmetro dos grafos de fluxo chamado *profundidade*. Daí em diante, vale a pena ter em mente a profundidade para um grafo de fluxo bem estruturado, pois a profundidade é o maior número de lados retrocedentes em qualquer percurso sem ciclos.

Exemplo 10.11. Na Fig. 10.16, a profundidade é 3, porque existe o percurso

$$[1, 2, 3, 2, 4, 5, 2]$$

com três lados retrocedentes, mas nenhum percurso sem ciclos com mais de três lados retrocedentes. Para ver isso, observe que qualquer percurso sem ciclos "mais profundo" possível somente ladrilhos retrocedentes. Em geral, podemos ter uma mistura de lados retrocedentes, progressivos e cruzados em percursos mais profundos. □

Podemos provar que a profundidade de um grafo é menor do que o que se pode alcançar uniformemente de profundidade de um ladrilhamento de um plano num grafo de fluxo. Se um grafo de fluxo reduzível pudesse substituir "retrocedente" por "retorno" na definição de "profundidade", uma vez que os lados retrocedentes em qualquer supressão exaustiva de lados retrocedentes. A noção de profundidade se torna independente da sequência exaustiva escolhida.

Intervalos

A decisão de um grafo de fluxo em intervalos serve para coletar informações estruturais sobre um grafo de fluxo. A estrutura por sua vez permite nos aplicar as regras para a análise de fluxo de dados descritas em seções subsequentes, começando na Seção 10.5.

Intuitivamente, um "intervalo" num grafo de fluxo é uma região natural na qual qualquer estrutura de laço que se pode inferir nos moldes da Seção 8.10 tem propriedade importante: uma de seus intervalos é que possui um nó de cabeçalho que domina todos os nós no intervalo, isto é, cada caminho de entrada para a região passa pelo cabeçalho. Cada grafo de fluxo G com nó inicial a_0 e um nó n de G é *decomposto em cabeçalho* e denotado por $\text{Int}_G(n)$ e depende dos nós e arcos a seguir:

1. n está em $\text{Int}_G(n)$.

2. Se todos os predecessores de algum nó $m \neq a_0$ estão em $\text{Int}_G(n)$, então m está em $\text{Int}_G(n)$.

3. Nada mais está em $\text{Int}_G(n)$.

Podemos, por conseguinte, construir $\text{Int}_G(n)$ começando por n e acrescentando nós por meio da regra (2). Note importante que, ao aplicarmos a regra (2), não precisamos que todos os predecessores de um nó sejam todos em $\text{Int}_G(n)$ para acrescentar o nó a $\text{Int}_G(n)$; basta que o nó seja eventualmente atingido pela regra (2). Eventualmente, não poderemos acrescentar mais nós a $\text{Int}_G(n)$, e o conjunto resultante de nós é o intervalo com cabeçalho n.

Partições de Intervalos

Dado qualquer grafo de fluxo G, podemos particionar G em intervalos disjuntos como segue.

Algoritmo 10.15 Análise de intervalos de um grafo de fluxo.

Entrada. Um grafo de fluxo G com nó inicial a_0.

Saída. Uma partição de G num conjunto de intervalos disjuntos.

Método. Para qualquer nó n, computamos $\text{Int}_G(n)$ pelo método delineado acima:

```
h(n) = {n};
enquanto existir um nó m ≠ n,
        do qual todos os predecessores estejam em h(n) faça
    h(n) = h(n) ∪ {m}
```

Há três possibilidades que são verdadeiras sobre os intervalos na partição assim obtida, como segue. Inicialmente, nenhum nó é "selecionado".

Assumimos, neste exemplo, que os lados de um nó são ordenados de modo que a seja visitado antes de b se $a < b$.

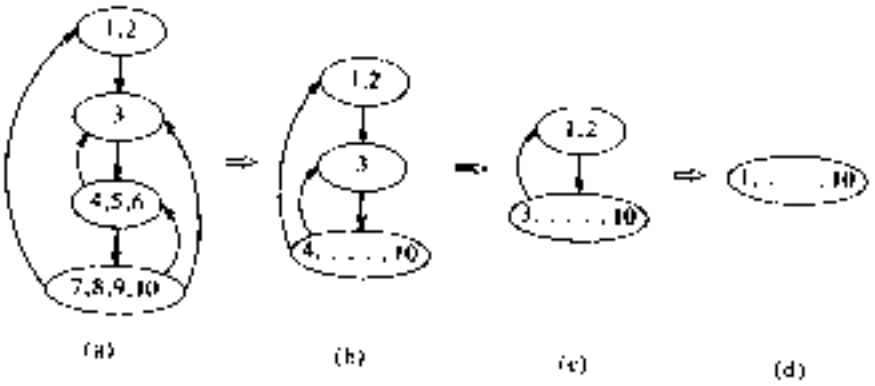


Fig. 10.45. Seqüência de grafos de intervalos

```
Construir I(r) = {selecionar todos os nós nesse intervalo}
Enquanto existir um nó n ainda não "selecionado",
    e cujo(s) predecessor(es) já tenham sido "selecionado(s)" faça
    construir I(n) = {n "selecionado" e todos os nós ainda não "selecionados"}
```

Uma vez que todos os predecessores de n estejam selecionados, n jamais poderia ser adicionado a algum intervalo que não contenha n. Por conseguinte, os candidatos a serem incluídos em $I(n)$ são aqueles ainda não selecionados para pertencerem a outras partições. Consequentemente, $I(n)$ é o intervalo com cabeçalho n. Os intervalos são criados no Algoritmo 10.15 não através de uma varredura física em intervalos. Igualmente, na medida em que todos os nós são alcançáveis a partir de r, pode ser provado por indução sobre o tamanho do percurso a partir de r para n, que n será eventualmente selecionado e incluído no intervalo de algum cabeçalho e se tornará cabeçalho de seu próprio intervalo, mas não antes de todas as suas arestas de entrada terem sido examinadas. Por conseguinte, o conjunto de intervalos construído pelo Algoritmo 10.15 particiona verdadeiramente G.

Exemplo 10.34. Vamos ilustrar a execução do intervalo para a Fig. 10.45. Os nós são numerados em ordem de visita em profundidade. Podemos adicionar 2 a $I(1)$ porque 1 é o único predecessor de 2 e $I(1) = \{1\}$. Não podemos adicionar 3 a $I(1)$ porque 3 também possui predecessores de 8, que não estão em $I(1)$. Similarmente, não podemos adicionar 4, 5 ou 6 a $I(1)$, porque esses nós possuem predecessores em 7 e 2, pois ainda não foram incluídos em $I(1)$. Por conseguinte, $I(1) = \{1, 2\}$.

Podemos adicionar 3 a $I(3)$ porque 3 possui alguns predecessores "selecionados", 1 e 2, mas o próprio 3 não está num intervalo. Similarmente, também podemos adicionar 4 a $I(3)$, e desse modo $I(3) = \{3, 4\}$. Agora, 4 é um cabeçalho porque possui um predecessor, 3, num intervalo. Podemos adicionar 5 e 6 a $I(4)$, porque esses nós somente possuem como predecessor, nós que já foram selecionados, por exemplo, 7 possui predecessor 10.

Na segunda, 7 se torna um cabeçalho e podemos adicionar 8 a $I(7)$. Na segunda, podemos adicionar 9 e 10, porque esses nós somente possuem 8 como predecessor. Consequentemente, os intervalos na partição da Fig. 10.45 são

$$I(1) = \{1, 2\}; \qquad I(4) = \{4, 5, 6\}$$
$$I(3) = \{3\} \qquad I(7) = \{7, 8, 9, 10\}$$

Grafos de Intervalos

A partir dos intervalos de um grafo de fluxo G, podemos construir um novo grafo de fluxo $I(G)$ pelas seguintes regras:

1. Os nós de $I(G)$ correspondem aos intervalos da partição de intervalos de G.

2. O nó inicial de $I(G)$ é o intervalo de $I(r)$ que contém o nó inicial de G.

3. Existirá uma aresta da parte no intervalo $I(n)$ para um intervalo diferente, $I(h)$, se somente se G existir uma aresta de algum nó em $I(n)$ para o cabeçalho de $I(h)$. Notemos que não poderia haver mais de uma aresta em $I(G)$ de algum nó de $I(n)$ de forma G que não modela a execução, porque não haveria forma pela qual o predicado seria adicionado a $I(n)$ no Algoritmo 10.15.

Podemos aplicar o Algoritmo 10.15 a uma construção de grafos de intervalos alternativamente, produzindo a sequência de grafos $G, I(G), I(I(G)), \ldots$. Eventualmente, a seqüência atinge um grafo em que cada um dos nós seja um intervalo em si. Esses grafos são chamados de *grafos de fluxo limite* de G. É um fato interessante que um grafo de fluxo seja redutível se e somente se seu grafo de fluxo limite for um único nó.

Exemplo 10.35. A Fig. 10.46 mostra o resultado da aplicação da construção de intervalos repetidamente à Fig. 10.45. Os intervalos que pode-riam formar foram escolhidos no Exemplo 10.34, e o grafo de intervalos construído a partir do nosso exemplo é o da Fig. 10.46(a). Notemos que o nó $I(1) = 7$ na Fig. 10.46(a) não acarreta um laço para nós do tipo que representa $I(7,8,9,10)$ por si mesmo na Fig. 10.45(b), porque a construção de grafos de intervalos exclui explicitamente tais laços. Notemos também que o grafo de fluxo da Fig. 10.45 é redutível porque seu grafo de fluxo limite é um único nó. □

Divisão de Nós

Se um grafo de fluxo limite que seja algo mais do que um único nó, podemos prosseguir aplicando somente as duas construções anteriores. Se um nó n possui k predecessores, podemos substituir n por k nós, $n_1, n_2, \ldots, n_k$. O i-ésimo predecessor de n se torna o predecessor de n_i somente, enquanto todos os sucessores de n se tornam sucessores de todos os n_i's.

Se aplicarmos o Algoritmo 10.15 ao grafo resultante, cada nó possuirá um único predecessor e, consequentemente, irá eventualmente se tornar parte do intervalo do predecessor. Por conseguinte, uma seqüência de nós mais modesta de partições em intervalos, resultando em grafos com um número menor de nós. Como consequência, a construção de grafos de intervalos, entre nenhum, quando necessária, uma divisão de nós, tem a vantagem de que atinge um grafo com um único nó. O segredo dessa observação irá se tornar claro na próxima seção, quando projetarmos algoritmos de análise de fluxo de dados que são dirigidos por essas duas operações sobre grafos.

Exemplo 10.36. Consideremos mais uma vez o grafo de fluxo da Fig. 10.9(a), igual ao seu próprio grafo de fluxo limite. Podemos dividir o nó 2 em 2_1 e 2_2

---

[1] Este fato é discutido em detalhes em [Aho e Ullman (1977)].

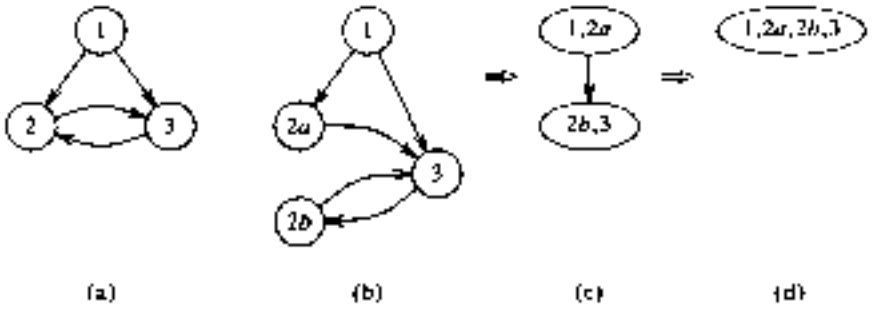


Fig. 10.49. Derivada de dois seguidos por partição em intervalos.

2b, com predecessores 1 e 3, respectivamente. Este grafo é mostrado na Fig. 10.49(b). Se aplicarmos o particionamento de intervalos duas vezes, obtemos a sequência de grafos mostrada na Fig. 10.49(c) e (d), ficando a único nó. □

Análise $T_1 - T_2$

Uma forma conveniente de se alcançar o mesmo efeito que a análise de intervalos é aplicar duas transformações simples aos grafos de fluxo:

T_1. Se a é um nó com um laço, isto é, um lado $a \to a$, remover esse lado.

T_2. Se existir um nó b que não seja inicial, que tenha um único predecessor a, então a pode ser unido a b através da remoção de a e fazendo-se todos os sucessores de b (se houver) possivelmente serem sucessores de a.

Alguns fatos interessantes a respeito das transformações T_1 e T_2 são:

1. Se aplicarmos T_1 e T_2 a um grafo de fluxo G em qualquer ordem até que resulte um grafo de fluxo para o qual nenhuma aplicação de T_1 ou de T_2 seja possível, um grafo de fluxo único resulta. A razão está em que todo candidato para remoção de laço por T_1 continua por T_2 se mantem um candidato, mesmo que alguma outra aplicação de T_1 ou de T_2 seja a primeira.

2. O grafo de fluxo resultante da aplicação exaustiva de T_1 e de T_2 a G é o grafo de fluxo limite de G. A prova é um tanto sutil e é deixada como exercício. Como consequência, uma outra definição de "grafo de fluxo redutível" é que é aquele que pode ser convertido a um único nó por T_1 e T_2.

Exemplo 10.37. Na Fig. 10.50 vemos uma sequência de reduções T_1, e T_2, começando com o grafo de fluxo que é um terceiro exemplo da Fig. 10.49(b). Na Fig. 10.50(b), o consumido d. Notemos que o laço em c na Fig. 10.50(b) resulta do lado $d \to c$ na Fig. 10.50(a). O laço é removido por T_1 na Fig. 10.50(c). Notemos, igualmente, que quando c é removido na Fig. 10.50(d), os lados provenientes de c e b para cd, e se tornam um único lado. □

Regiões

Relembremos da Seção 10.5 que uma região em um grafo de fluxo é um conjunto de nós N que inclui um cabeçalho, que domina todos os outros nós na região. Todos os lados correntes em N estão na região exceto (possivelmente) para alguns daqueles que estejam no cabeçalho. Por exemplo, cada intervalo é uma região, mas existem regiões que não são intervalos porque, por exemplo, podem conter alguns nós que um intervalo de mesma cabeça não poderia conter (alguns lados de volta para o cabeçalho). Existem, também, regiões menores do que qualquer intervalo como iremos ver.

A medida que reduzimos um grafo de fluxo através de T_1 e T_2, todo instante as seguintes condições são verdadeiras:

1. Um nó representa uma região de G.

2. Um lado de a até b representa um conjunto de lados. Cada um destes lados vai de algum nó da região representada por a até o cabeçalho da região representada por b.

3. Cada nó e lado de G é representado por exatamente um nó ou lado do grafo corrente.

Para vermos por que essas observações são válidas, notemos primeiro que as mesmas seguram trivialmente para o próprio G. Cada nó é uma região por si mesmo e cada lado representa somente a si mesmo. Suponhamos que apliquemos T_1 a algum nó a representando uma região R, enquanto o laço $a \to a$ representa algum conjunto de lados E, todos os quais necessariamente tem que entrar no cabeçalho de R. Se removermos os lados E a região R é ainda uma região e, por consequinte, após haver removido o lado $a \to a$, o nó a representa R e os lados de E, o que preserva as condições (1) e (3) acima.

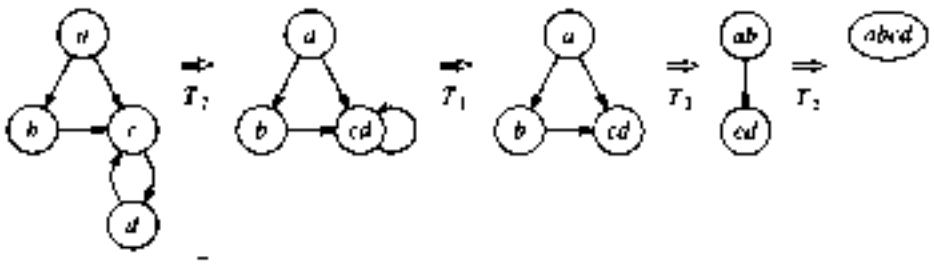


Fig. 10.50. Reduções através de T_1 e T_2.

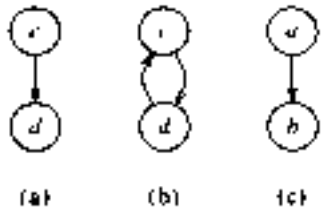


Fig. 10.51. Algumas regiões.

```
(1)  Dom(s) = {n0};
(2)  para n em N − {n0} faça Dom(n) = N
       − fim de inicialização −
(3)  enquanto ocorrer mudança a algum Dom faça
(4)      para n em N − {n0} faça
(5)          Dom(n) = {n} ∪ (∩ Dom(p) sobre todos os predecessores p de n)
```

Fig. 10.52. Algoritmo de cômputo dos dominadores

Se, no lugar, usarmos T_2 para consumir o nó b pelo nó a, faça $a+b$ representar as regiões R e S respectivamente. Igualmente, seja L o conjunto de lados representados pelo lado $a \rightarrow b$. Afirmamos que R, S e L juntos formam uma região cujo cabeçalho é h (o mesmo cada uma em S). Sendo assim, deverá haver algum percurso para o cabeçalho de R que não termine com algum lado de L. Então, o último lado desse percurso terá de ser representado no grafo de fluxo corrente por algum arco cuja cabeça seja b. Mas não poderá haver um tal lado, ou T_2 não poderá ser usada para consumir b.

Exemplo 10.38. O nó resultado a na Fig. 10.50(b) representa a região mostrada na Fig. 10.51(a), a qual foi formada ao ter consumido d. Sabemos que o lado $d \rightarrow c$ faz parte da região na Fig. 10.51(b), pois esse lado é representado pelo laço em c. No entanto, na Fig. 10.50(c), o lado $d \rightarrow c$ foi removido e o nó a agora representa a região mostrada na Fig. 10.51(b).

Na Fig. 10.50(d), o nó a ainda representa a região da Fig. 10.51(b), enquanto que o nó b representa a região da Fig. 10.51(c). O lado $a \rightarrow b$ na Fig. 10.50(d) representa os lados $a \rightarrow c$ e $c \rightarrow b$ do grafo de fluxo original na Fig. 10.50(a). Ao aplicarmos T_1 a qualquer um dos nós na Fig. 10.50(e), o nó restante representa todo o grafo de fluxo, a Fig. 10.50(a). □

Deveríamos observar que a propriedade de redução de T_1 e T_2 mencionada acima também é útil para a análise de intervalos. Deixamos como exercício o fato de que, à medida que constituímos $\mathrm{I}(G)$, $\|\mathrm{I}(G)\|$, etc., sempre, cada nó em cada um desses grafos representa uma região e cada lado, um conjunto de lados satisfazendo a propriedade 12 acima.

Encontrando os Dominadores

Encerramos esta seção com um algoritmo eficiente para um conceito que temos usado frequentemente, e que continuaremos usando, no desenvolvimento da teoria dos grafos de fluxo e análise de fluxo de dados. Daremos um algoritmo simples para computar os dominadores de cada nó num grafo de fluxo, baseados no princípio de que, se $p_1, p_2, \ldots, p_k$ são todos predecessores de n e $d \neq n$, então d domina n se e somente se d domina p_i para cada i. O método é semelhante à análise de fluxo de dados para diante, tendo a interseção como operador de confluência (por exemplo, expressões disponíveis), na qual iniciamos uma aproximação do conjunto de dominadores de n e o refinamos repetidamente, visitando todos os nós a cada vez.

Neste caso, a aproximação inicial que escolhemos para o nó inicial n_0 é $\{n_0\}$, pois n_0 é dominado somente por n_0. Para todos os outros nós n, iniciamos com o conjunto de todos os nós, pois sabemos que os dominadores de n devem estar entre esses nós, e começamos com a suposição mais otimista possível, a saber, que n é dominado por todos eles. Em seguida, iteramos sobre todos os nós, exceto n_0, refinando o conjunto de dominadores de n a cada passagem, até que nenhuma mudança ocorra.

Algoritmo 10.16 Encontrar todos os dominadores.

Entrada. Um grafo de fluxo G com conjunto de nós N, conjunto de lados E e nó inicial n_0.

Saída. A relação *domina*.

Método. Computamos $Dom(n)$, o conjunto de dominadores de n, iterativamente, pelo procedimento da Fig. 10.52. Ao final, d está em $Dom(n)$ se e somente se d *domina* n. O leitor pode conhecer os detalhes relacionados a como as mudanças em $Dom(n)$ são detectadas; o Algoritmo 10.2 servirá como modelo.

Pode-se mostrar que $Dom(n)$ computado à linha (5) da Fig. 10.52 é sempre um subconjunto do $Dom(n)$ corrente. Como Dom não pode diminuir indefinidamente, terminaremos eventualmente o laço **enquanto**. A prova que, após a convergência, $Dom(n)$ é o conjunto de dominadores de n é deixada para o leitor interessado. O algoritmo da Fig. 10.52 é razoavelmente eficiente, na medida em que Dom possa ser representado por uma versão de bits e as operações sobre conjuntos da linha (5) possam ser feitas sem muito custo lógico. □

Exemplo 10.39. Vamos retomar o grafo de fluxo da Fig. 10.45 e supenhamos que no laço **para** da linha (4) os nós sejam visitados em ordem numérica. O nó 2 possui somente 1 como predecessor, dessa forma $Dom(2) = \{2\} \cup Dom(1)$. Como 1 é o nó inicial, $Dom(1)$ foi atribuído $\{1\}$ a linha (1). Por conseguinte, $Dom(2)$ é estabelecido com $\{1,2\}$ na linha (5).

Em seguida, o nó 3 com predecessores 1, 2, 4, e 8 é considerado. A linha (5) nos dá $Dom(3) = \{3\} \cup (\{1\} \cap \{1,2\} \cap \{1,2,\ldots,15\}) = \{1,3\}$. Os conjuntos restantes são

$$
\begin{aligned}
Dom(4) &= \{4\} \cup (Dom(3) \cap Dom(7)) = \{4\} \cup (\{1,3\} \cap \{1,2,\ldots,6\}) = \{1,3,4\}\\
Dom(5) &= \{5\} \cup Dom(4) = \{5\} \cup \{1,3,4\} = \{1,3,4,5\}\\
Dom(6) &= \{6\} \cup (Dom(4) - \{6\}) \cup \{\ldots,3,4\} = \{1,3,4,6\}\\
Dom(7) &= \{7\} \cup (Dom(5) \cap Dom(6) \cap Dom(10))\\
&= \{7\} \cup (\{1,3,4,5\} \cap \{1,3,4,6\} \cap \{1,2,\ldots,10\}) = \{1,3,4,7\}\\
Dom(8) &= \{8\} \cup Dom(7) = \{8\} \cup \{1,3,4,7\} = \{1,3,4,7,8\}\\
Dom(9) &= \{9\} \cup Dom(8) = \{9\} \cup \{1,3,4,7,8\} = \{1,3,4,7,8,9\}\\
Dom(10) &= \{10\} \cup Dom(8) = \{10\} \cup \{1,3,4,7,8\} = \{1,3,4,7,8,10\}
\end{aligned}
$$

A segunda passagem através do laço **enquanto** não produz mudanças, de forma que os valores acima produzem a relação *domina*. □

10.10 ALGORITMOS DE FLUXO DE DADOS EFICIENTES

Nesta seção, consideraremos duas formas de usar a teoria dos grafos de fluxo para acelerar a análise de fluxo de dados. A primeira é uma aplicação do ordenamento em profundidade, a fim de reduzir o número de passagens que os algoritmos iterativos da Seção 10.6 fazem e a segunda usa intervalos ou as transformações T_1 e T_2 para generalizar a abordagem dirigida pela sintaxe da Seção 10.5.

Ordenamento em Profundidade nos Algoritmos Iterativos

Em todos os problemas estudados até então, tais como as definições recidentes, expressões disponíveis ou variáveis vivas, qualquer evento

de significância, para um nó será propagado através de um arco ao longo de um percurso acíclico. Por exemplo, se uma definição d gera uma variável v no bloco B, então existe algum percurso acíclico a partir do bloco que contém d até B tal que todas as definições de variáveis diferentes de v ao longo daquele percurso são anuladas. Similarmente, se uma expressão e não está disponível na entrada para um bloco B, então, algum percurso acíclico a partir do bloco que demonstra essa fato, ou o próprio ponto de um nó inicial e não contém nenhum nó que mate ou gere e, ou um percurso cíclico e de um bloco que mate e e ao longo do percurso não existe geração subsequente de e. Finalmente, para as variáveis vivas, se uma variável v está viva à saída do bloco B, então existe um percurso acíclico a partir de B até um bloco ao longo do qual não existem definições de v.

O leitor deverá ser capaz, em cada uma das classes gerais de percursos definidos anteriormente, de encontrar algoritmos de fluxo de dados que resolvem o problema da propagação na direção para frente. Para problemas de variáveis vivas, onde propagamos pedaços a mesma média de certas passagens pode ser atingida se escolhermos a ordem certa em profundidade. Por conseguinte, podemos propagar o uso de uma variável no Bloco 17 de volta ao longo do percurso

$$5 \to 8 \to 17 \to 18 \to 16 \to 23 \to 15 \to 4 \to 10 \to 13$$

em uma passagem para o nó 4, onde precisamos esperar pela próxima passagem para atingir 8. Na segunda passagem, o nó 8 atinge o nó 4, e o nó 4 atinge, via de saída 0, para o nó 3. Em geral, um número de passagens igual à profundidade mais um é suficiente para estabelecer o que queremos saber para um laço ao longo de qualquer percurso acíclico, se escolhermos a ordem em profundidade para visitar os nós em uma passagem, porque então os usos se propagarão ao longo de qualquer sequência decrescente em uma única passagem.

Análise de Fluxo de Dados Baseada em Estruturas

Como já pelo menos dezenas de vezes, pretendemos implementar algoritmos que visitam os nós e aplicam funções de fluxo de dados um número de vezes não maior do que a profundidade de aninhamento do grafo de fluxo e frequentemente o caso médio será o dobro ou um número de vezes ainda menor do que esse número. Se o leitor quer ver a prova de um verdadeiro limite de tempo, ainda não foi mencionado o suficiente mas também é composto, baseado na análise de intervalos, tem sido usado em diversos compiladores. Sinceramente, as ideias expostas aqui se aplicam aos algoritmos de fluxo de dados dirigidos pela sintaxe para todos os tipos de enunciados de controle estruturados, não somente `if-then-else`, `do`, `while`, discutidos na Seção 10.5, sendo que esses algoritmos também aparecerão em várias outras ocasiões.

Iremos buscar nossos ligeiramente estrutura mais rica de nós grafos de fluxo pelas transformações $T1$ e $T2$. Como na Seção 10.5, estamos envolvidos com as definições que são geradas e mortas à medida que caminhamos em um arco de uma região. Diferentemente das regiões definidas pelos enunciados `if-else` e `while`, uma região geral pode ter múltiplas saídas, de forma que para cada bloco B em uma região R iremos computar conjuntos gerados e_gen_B e mortos e_kill_B das definições geradas e mortas respectivamente, ao longo de certos caminhos dentro da região a partir do cabeçalho até o final do bloco B. Esses conjuntos serão usados para determinar, de maneira análoga à Seção 9.3, quais definições para qualquer conjunto S de definições, que certamente atinge o final do bloco B viajando ao longo de percursos completamente dentro de R, desde que todas as somente as definições em S atinjam o cabeçalho de R.

Como ilustrado nas Seções 10.5 e 10.6, as definições incidentes no final do Bloco B caem em duas classes.

1. Aquelas que são geradas dentro de R e se propagam até o final de B independentemente de S.

2. Aquelas que são geradas em R mas, também, podem atingir o final de B ao longo de algum percurso a partir do cabeçalho de R até o final de B, e, por conseguinte, estão em $gen_B(S)$ se e somente se estiverem em S.

Note-se que [illegible] pode ser [illegible] conjunto [illegible] ao longo

Consequentemente, podemos escrever essa forma

$$gerada_{ent}(B) = gerada_{ent}(B_1) \cup (V - anula_{ent}(B_1)) \cdot gerada_{ent}(B)$$

Observação análoga é válida para se computar $anula_{ent}(B)$. Para regiões progressivas, quaisquer duas delas definidas por alguma decomposição (T, I) de um grafo de fluxo. Para o caso ilustrado, assumimos que o grafo de fluxo se constitui de apenas duas regiões, mas a modificação permite que o algoritmo funcione para grafos quaisquer, igualmente.

Vamos supor que B seja uma região constituída de um único bloco, B. Aqui a função de transferência da região é a função de transferência do bloco, ou seja, uma definição atinge o início de um bloco se e somente se ela ocorre no bloco ou é gerada pelo bloco e vai até o seu conjunto G mais o conjunto I desse bloco.

$$gerada_{ent}(B) = gerada(B)$$
$$anula_{ent}(B) = anula(B)$$

Vamos agora considerar a construção de uma região B por I_2, isto é, B é formada quando B_1 e somente B_1 constituem B, como ilustrado na Fig. 10.53. Primeiro, notemos que dentro da região B não existem laços em B de volta à B, já que qualquer laço partindo de B_2 para o cabeçalho de B não é parte de B. Por conseguinte, qualquer percurso totalmente dentro de B passa tipicamente através de B_1 primeiro e, então, através de B_2 opcionalmente, mas não pode retornar a B_1. Notemos igualmente que o cabeçalho de B é o cabeçalho de B_1. Podemos concluir que dentro de B, B_2 não afeta a função de transferência desse B, isto é,

$$gerada_{ent}(B) = gerada_{ent}(B_1)$$
$$anula_{ent}(B) = anula_{ent}(B_1)$$

para todo B em B.

Para B em B_2, uma definição pode atingir o final de B se qualquer uma das condições seguintes estiver em vigor:

1. A definição é gerada dentro de B_1.
2. A definição é gerada dentro de B_2, atinge o fim de algum predecessor do cabeçalho de B_2 e não é morta ao longo do cabeçalho de B_2 para B.
3. A definição está no conjunto I do cabeçalho de B_2, não sendo morta ao longo de algum predecessor do cabeçalho de B_2 e não sendo morta ao longo do cabeçalho de B_2 para B.

Por conseguinte, as definições que atingem o fim daqueles blocos em B que sejam predecessores do cabeçalho de B_2 desempenham um papel especial. Para esclarecer, vamos supor que S seja o conjunto de definições que entram no cabeçalho de B_2, a medida que percorremos todos os caminhos que atingem o cabeçalho de B_2 através de um de seus predecessores. O conjunto de definições que atingem os predecessores do cabeçalho de B_2 se torna o conjunto de entrada para B_2 e aplicamos as funções de transferência para B_2, àquele conjunto.

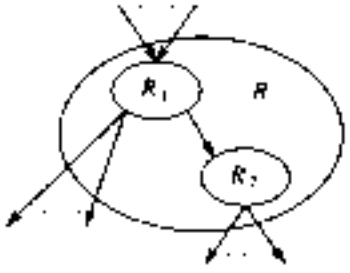


Fig. 10.53. Construção de B por I_2.

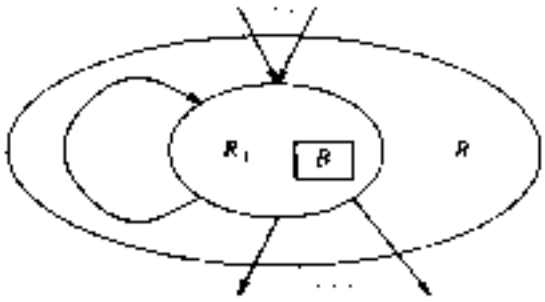


Fig. 10.54. Construção de região B através de I_2.

Por conseguinte, seja A a união de todos os $gerados_{ent}(P)$ para todos os predecessores P do cabeçalho de B, e seja M a interseção de todos os $anulados_{ent}(P)$ para todos os predecessores P. Então, seja V o conjunto de definições que atingem o cabeçalho de B, o conjunto de definições que atingem o cabeçalho de B_1 ao longo de percursos totalmente dentro de B é $(A \cup (V - M))$. Consequentemente, a função de transferência em B para aqueles blocos B em B_1 pode ser computada por

$$gerada_{ent}(B) = gerada_{ent}(B_1) \cup (A \cup (V - M)) \cdot gerada_{ent}(B)$$
$$anulada_{ent}(B) = anulada_{ent}(B_1) \cup M \cdot anulada_{ent}(B)$$

Consideramos, em seguida, o caso ocorrido quando uma região B é constituída a partir de uma região B_1 através da transformação I_1. A situação geral é mostrada na Fig. 10.54, podemos que B consiste em B_1 mais todos os laços refluentes para o cabeçalho de B. Um laço também é cabeçalho de B, naturalmente. A propriedade crucial é que cabeçalho de uma região deve ser alcançado, como ponto de entrada, somente, não importa se considerado. Por conseguinte, todas as definições geradas ao final de um bloco B são geradas em uma de duas formas:

1. A definição é gerada dentro de B_1 e não precisa existir nos laços refluentes incorporados em B a fim de atingir o fim de B.
2. A definição é gerada em algum lugar dentro de B, atinge um predecessor do cabeçalho, segue um laço refluente e não é morta ao longo do cabeçalho para B.

Se fizermos G ser a união de $gerados_{ent}$ para todos os predecessores do cabeçalho em B_1, então,

$$gerada_{ent}(B) = gerada_{ent}(B_1) \cup (G - anulada_{ent}(B_1))$$

Uma definição é morta ao longo de cabeçalho para B se e somente se for morta ao longo de todos os percursos de cabeça, excluindo quaisquer laços refluentes incorporados em B não fazendo com que mais definições sejam mortas. Isto é,

$$anulada_{ent}(B) = anulada_{ent}(B_1)$$

Exemplo 10.40. Vamos reconsiderar o grafo de fluxo da Fig. 10.38, cuja decomposição (T, I) é mostrada na Fig. 10.55, onde as regiões da decomposição recebem nomes. Mostramos, também na Fig. 10.56, os vetores ger e $anul$ que representam os três definições e seus conjuntos $gerada$ e $anulada$ em cada um dos blocos da Fig. 10.55.

Começando de fora para dentro, notamos que para regiões de um único nó, que chamamos A, B, C e D, os valores de $gerada$ e $anulada$ são dados pela tabela na Fig. 10.56. Pode-nos, então, prosseguir até a região E, que é formada quando C e somente D entram de F. Seguindo as regras para I_1, acima, notamos que $gerada_{ent}$ e $anulada_{ent}$ não mudam para C isto é,

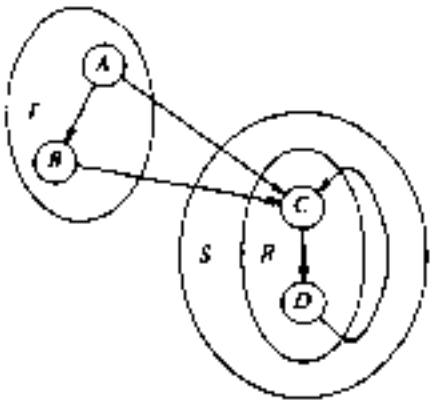


Fig. 10.55. Decomposição de um grafo de fluxo.

$$gerados_{C_1} = gerados_{T_1} = \{d3\}$$
$$mortos_{C_1} = mortos_{T_1} = \{d1, d2\}$$

Para o nó D, temos que encontrar na região C a união de *gerados* para todos os predecessores do cabeçalho da região D. Naturalmente, o cabeçalho da região D é o nó D e existe somente um predecessor daquele nó na região C, nomeadamente, o nó C por conseguinte,

$$gerados_{C_2} = gerados_{C_1} \cup (gerados_{C_1} - mortos_{C_1}) = \{d3\} \cup (\{d3\} - \{d1, d2\}) = \{d3\}$$
$$mortos_{C_2} = mortos_{C_1} \cup mortos_{C_1} \quad gerados_{C_1} = \{d3\} \cup \{d1, d2\} = \{d1, d2, d3\}$$

Consideramos, agora, a região S a partir da região R através de T_1. Os conjuntos operações mudam de forma que temos

$$mortos_{T_1} = mortos_{A,1} = \{d1\}$$
$$gerados_{T_1} = mortos_{A,2} = \{d2\}$$

Para computarmos os conjuntos *gerados* para S, notamos que o único lado reflexivo para o cabeçalho de S que é interposto inteiramente em R para S é o lado $D \rightarrow C$. Por conseguinte,

$$gerados_S = gerados_{T_1} \cup (gerados_{T_1} - mortos_{T_1}) = \{d3\} \cup (\{d3\} - \{d1\}) = \{d3\}$$
$$mortos_S = gerados_{T_1} \cup (gerados_{T_1} - mortos_{T_1}) = \{d1\} \cup (\{d3\} - \{d1\}) = \{d1, d3\}$$

O cômputo para a região T é análogo àquele para a região R e obtemos

$$gerados_{T_2} = \{d3\}$$
$$mortos_{T_2} = \{d1\}$$
$$gerados_{T_3} = \{d3\}$$
$$mortos_{T_3} = \{d1\}$$

BLOCO	*GERADAS*	*MORTAS*
A	{d3}	{d1}
B	{d3}	{d4}
C	{d3}	{d1}
D	{d4}	{d3}

Fig. 10.56. Informações de *gerados* e *mortos* para o fluxograma da Fig. 10.53.

Finalmente, computamos *gerados* e *mortos* para a região C todo o grafo de fluxo. Como C é construído quando T_2 consome a última transformação T, os valores de *gerados* e *mortos* para os nós A e B são aqueles que já foram fornecidos acima. Para C e D, notamos que o cabeçalho de S em C_1 possui dois predecessores na região T, nomeadamente, A e B. Por conseguinte, computamos

$$G = gerados_{T_2} \cup gerados_{T_3} = \{d3\}$$
$$M = mortos_{T_2} \cap mortos_{T_3} = \{d1\}$$

Podemos, então, computar

$$gerados_{C_3} = gerados_{C_2} \cup (G - mortos_{C_2}) = \{d3\}$$
$$mortos_{C_3} = mortos_{C_2} \cup M \quad gerados_{C_3} = \{d1, d2\}$$
$$gerados_{C_4} = gerados_{C_3} \cup (G - mortos_{C_3}) = \{d3\}$$
$$mortos_{C_4} = mortos_{C_3} \cup M = \{d1, d2\}$$

Todos os computados $gerados_{C_i}$, $mortos_{C_i}$ para cada bloco B, onde C é a região que corresponde ao grafo de fluxo, computamos essencialmente $saída[B]$ para cada bloco B. Isto é, se voltarmos à definição de $saída$, $saída[N] = gerados_{C_i} \cup (in_{C_i} - mortos_{C_i})$, notamos que in_{C_i} é o conjunto $saída[D]$. Mas, claro, $in_{C_i} = gerados_{C_3}$. Por conseguinte, a forma do algoritmo de definições incidentes baseado em estruturas consiste em usar os conjuntos $gerados$ e $saída$ e conjuntos $mortos$ conjuntos $gerados$ juntando a união dos conjuntos $saída$ dos predecessores. Esses passos são sumarizados no algoritmo seguinte.

Algoritmo 10.17 Definições incidentes baseadas em estruturas

Entrada. Um grafo de fluxo reduzido G e conjuntos de definições $gerados[B]$ e $mortos[B]$ para cada bloco B de G.

Saída. $Entrada[B]$ para cada bloco B

Método

1. Encontrar a decomposição $T_1, T_2, \ldots, T_n$ para G.
2. Para cada região R na decomposição, de cima para baixo, computar $gerados_{C_i}$, $mortos_{C_i}$ para cada bloco B em R.
3. Se C_n é o nome da região que consiste em todo o grafo, então para cada bloco B, $saída[B]$ igual à união sobre todos os predecessores P do bloco B, de $gerados_{C_n}$.

Algumas Acelerações do Algoritmo Baseado em Estruturas

Primeiro, note-se que se tivermos uma função de transferência $f(x) = M_f \cup x$, a função não é modificada se removermos de M_f alguns de membros de G. Consequentemente, ao aplicarmos T_2 em vez de usar as fórmulas

$$gerados_{C_3} = gerados_{C_2} \cup (G - mortos_{C_2})$$
$$mortos_{C_3} = mortos_{C_2} \cup M$$

podemos substituir a segunda por

$$mortos_{C_3} = mortos_{C_2} \cup (M - G)$$

economizando uma operação para cada bloco da região R_2.

Outra idéia útil é notar que o único instante em que aplicamos T_1 após termos primeiro consumido alguma região R, por R_1 existem alguns lados refletentes partindo de R para cabeçalho de R. Em lugar de primeiro fazer as mudanças em R_1 e R_2 devido à operação T_2, podemos combinar os dois conjuntos de mudanças se fizermos o seguinte.

1. Usando a regra T_2, computamos a nova função de transferência para aqueles nós em R que sejam predecessores do cabeçalho de R.

2. Usando a regra T_1, computamos a nova função de transferência para todos os nós de R.

3. Usando a regra T_2, computamos a nova função de transferência para todos os nós de R. Notemos que a realimentação devida à aplicação de T_1 atinge os predecessores de R e é passada a todos os nós de R pela regra T_2; não há necessidade de se aplicar a regra T_1 a R.

Tratando Grafos de Fluxo Não-Redutíveis

Se as reduções T_1-T_2 de um grafo para um grafo de fluxo limite que não seja um único nó, precisamos realizar uma divisão de nós. Dividir um nó de um grafo de fluxo limite corresponde a duplicar a região inteira representada por aquele nó. Por exemplo, na Fig. 10.57 sugerimos o efeito que a divisão de nós poderia ter sobre o grafo de fluxo original de nove nós que foi particionado por T_1 e T_2 em três regiões conectadas por alguns lados.

Como mencionado na seção anterior, através da alternância das divisões com as sequências de reduções, estaremos garantidos em reduzir o grafo de fluxo a um único nó. O resultado da divisão é que alguns nós do grafo original terão mais de uma cópia na região representada pelo grafo de um único nó. Podemos aplicar o Algoritmo 10.17 a esta região com pouca mudança. A única diferença é que, quando dividimos um nó, os conjuntos *gerado* e *morto* para os nós do grafo original na região representada pelo nó dividido precisam ser duplicados. Por exemplo, quaisquer que sejam os valores de *gerado* e *morto* para os nós na região de dois nós da Fig. 10.57 à esquerda, se tornam *gerados* e *mortos* para cada um dos nós correspondentes em ambas as regiões de dois nós à direita. No passo final, quando computamos os conjuntos *entrada* para todos os nós, aqueles nós do grafo original que têm vários representantes na região final possuem seus conjuntos *entrada* computados tomando-se a união dos conjuntos *entrada* de todos os seus representantes.

No pior caso, a divisão de nós poderia exponenciar o número total de nós representados por todas as regiões. Por conseguinte, se esperamos que muitos grafos de fluxo sejam não-redutíveis, provavelmente não deveremos usar métodos baseados em estruturas. Felizmente, os grafos de fluxo não-redutíveis são suficientemente raros de forma que podemos geralmente ignorar o custo da divisão de nós.

10.11 UMA FERRAMENTA PARA A ANÁLISE DE FLUXO DE DADOS

Como apontamos anteriormente, existem fortes similaridades entre os vários problemas de fluxo de dados estudados. As equações de fluxo de dados da Seção 10.6 foram vistas serem caracterizadas por:

1. A função de transferência usada, a qual em cada caso estudado era da forma $f(X) = A \cup (X - B)$. Por exemplo, A = *morto* e B = *gerado* para as definições incidentes.

2. O operador de confluência, o qual em todos os casos até então tem sido ou a união ou a interseção.

3. A direção de propagação da informação, para frente ou para trás.

Como essas distinções não são grandes, não deveria ser surpreendente que todos esses problemas pudessem ser tratados de forma unificada. Uma tal abordagem foi descrita em Kildall [1973] e uma ferramenta para simplificar a implementação de problemas de fluxo de dados foi implementada por Kildall e usada pelo mesmo em vários projetos de compiladores. Não foi vista em uso disseminado provavelmente porque a quantidade de trabalho economizada pelo sistema não era tão grande quanto a economizada por ferramentas como os geradores de analisadores sintáticos. No entanto, deveríamos estar informados do que pode ser feito não somente porque isso sugere uma simplificação para implementadores de compiladores otimizantes, mas também porque auxilia a unificar as várias idéias que examinamos nas seções deste capítulo. Sobretudo, esta seção sugere como podem ser desenvolvidas estratégias mais poderosas de análise de fluxo de dados, aquelas que providenciam informações mais precisas do que os algoritmos mencionados até então.

Estruturas de Análise de Fluxo de Dados

Iremos descrever as estruturas que modelam problemas de propagação para adiante. Se considerarmos somente a técnica de solução iterativa para os problemas de análise de fluxo de dados, a direção não faz diferença; podemos reverter a direção dos lados e fazer alguns ajustes menores para lidar com o nó inicial e então tratar o problema para trás como se fosse para adiante. Os algoritmos baseados em estruturas são um tanto diferentes; os problemas para adiante e para trás não são resolvidos exatamente da mesma forma, porque o reverso de um grafo de fluxo redutível não precisa ser redutível. No entanto, o tratamento de problemas para trás será deixado como exercício e trataremos somente os problemas para adiante.

Uma *estrutura de análise de fluxo de dados* consiste em:

1. Um conjunto V de valores a serem propagados. Os valores de *entrada* e *saída* são membros de V.

2. Um conjunto F de *funções de transferência* de V para V.

3. Uma operação binária de *confluência* ∧ sobre V, para representar o operador de confluência.

Exemplo 10.41. Para as definições incidentes, V consiste em todos os subconjuntos dos conjuntos de definições do programa. O conjunto F

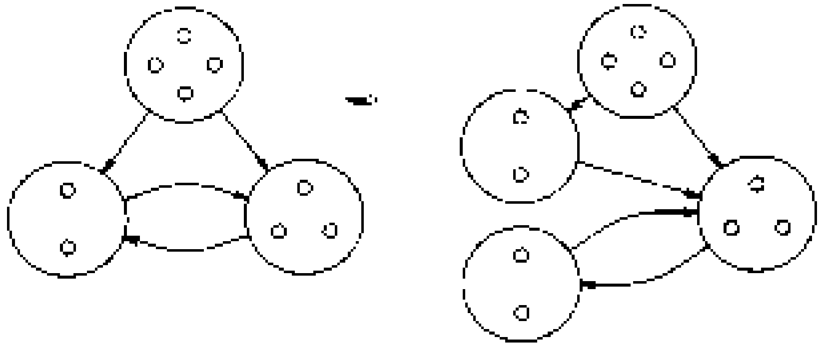


Fig. 10.57. Dividindo um grafo de fluxo não-redutível.

é o conjunto de todas as funções da forma $\lambda X.((X - A) \cup B)$, onde A e B são conjuntos de definições sobre V, e dizemos que A e B são os conjuntos de definições *mortas* e *geradas*, respectivamente. Finalmente, a ordem parcial $\sqsubseteq$ é $\subseteq$.

Para as expressões disponíveis, L consiste em todos os subconjuntos de expressões computadas pelo programa e F é o conjunto de expressões da mesma forma que acima, isto é, todas as $\lambda X.((X - A) \cup B)$, onde A e B são conjuntos de expressões. A operação de reunião é a interseção, naturalmente. ⊡

Exemplo 10.42. A abordagem de Kildall não é limitada a exemplos simples como os que temos até agora. Apesar da complexidade aparente, tudo o que temos de fazer é computar um ponto fixo de uma função monotônica. Os exemplos sugerem um exemplo muito mais poderoso, onde as informações concretas de fluxo de dados podem, na verdade, ser expressões sobre todos os pares de expressões que têm mesmo valor em um ponto. No entanto, temos sempre o pequeno sabor desse exemplo lançando mão de um método para informar que variáveis têm valores constantes, de uma forma que capturamos informações de propagação de constantes. Nossa nova constante compreende, por exemplo, que, quando x é definida como $a + b$ e a e b têm cada um valor constante, então x também tem uma constante de valor. Não podemos fazer isso somente com uma abordagem de propagação de constantes. ⊡

Em síntese, usamos as definições dependentes para a propagação de constantes, verificando que o enunciado d em uma possível definição de x não é $x := \ldots$, e, portanto, que x não tinha um valor constante. Naturalmente, em uma passagem, o lado direito de d não pode ser substituído por uma constante, e assim outra rodada de propagação de constantes detectaria que os usos de x definidos em d são efetivamente usos de uma constante.

Nossa estrutura no momento verifica o ponto de todos os mapeamentos a partir das variáveis de programa num conjunto particular de valores. Esse conjunto de valores consiste em

1. Todas as constantes.
2. O valor *não-constante*, o qual significa que a variável em questão foi determinada como não tendo um valor constante. Observe não constante, e não que ela seja atribuída a uma variável $\ldots$ se durante a análise de fluxo de dados descobrirmos dois percursos ao longo dos quais x tem 2 e 3, respectivamente, associados a ela ou um percurso ao longo do qual a definição pode ser feita de x fosse uma constante de teste.
3. O valor *indefinido*, que significa que pode ser afirmado a respeito da variável em questão, ou porque ela ainda não foi descoberta pela execução da análise de fluxo de dados ou porque nenhuma definição da variável que atinge esse ponto em questão.

Notamos que *não-constante* e *indefinido* não são os mesmos valores sob essa nova interpretação. O primeiro diz que a variável tem uma forma sob a qual uma variável pode ser definida que sabemos não ser uma constante, mas o segundo diz que sabemos muito pouco a respeito de uma variável que, de fato, não podemos dizer nada...

A operação de reunião é definida pela seguinte tabela. Fazemos a interseção dos membros de V isto é, um dos quais é *não-constante*, uma constante, ou *indefinido*, respectivamente. A operação $\sqcap p = \mu_1 \sqcap \mu_2$ é definida, de forma na Fig. 10.58, onde damos o valor de μ em termos dos valores de μ_1 e de μ_2, para cada variável v. Naquela tabela, c é uma constante arbitrária e d uma outra constante, ou outro elemento diferente de c. Por exemplo, se $\mu_1(v) = c$ e $\mu_2(v) = d$, uma constante diferente, aparentemente $\ldots$ temos os dois valores de v ao longo de dois caminhos percorridos até o ponto p, e daqueles percursos $\ldots$ não possui valor constante. Daí a escolha $\mu(v) = \text{não-constante}$. Como outro exemplo, se ao longo de pelo menos um percurso v é conhecido a respeito de μ_1 e μ_2 por $\mu_1(v) = \mu_2(v) = c$, então $\mu(v) = c$, ao longo de um ponto percorrido, e indefinido $\ldots$ Se v é definido como $a + b$, e a e b têm cada um valores constantes, digamos c e d, respectivamente, então v também tem um valor constante, a saber, $c + d$. Mas se a ou b não tiver valor constante, então v também não terá valor constante.

Finalmente, precisamos projetar os conjuntos de funções F que refletem a transferência de informações de um ponto para o final de um bloco. A descrição desse conjunto de funções é complicada, mas as ideias serão diretas. Iremos pelo conveniente fornecer uma base para o conjunto de funções através da descrição das funções que representam conjuntos singelos e um conjunto qualquer de funções poderá ser então construído através da composição de funções, a partir desse conjunto base de forma a refletir os blocos com mais de um enunciado de definição.

1. A função identidade está em F, esta função reflete uma região que não contém enunciados de definição. Se I é uma coleção de todas as variáveis para os valores, então $I\,\mu = \mu$. Notamos que μ pode não possuir a identidade para todas as variáveis.
2. Para cada variável v e constante c existe uma função $const(v, c)$ em F tal que, para cada mapeamento μ em V, temos $(const(v, c))\,\mu = \mu'$, onde $\mu'(v) = c$ e $\mu'(w) = \mu(w)$ para todas as variáveis w diferentes de v. Essas funções refletem a ação de um enunciado de atribuição $v := c$.
3. Para cada v e conjunto $v_1, v_2, \ldots, v_n$ existe uma função $sum(v, v_1, v_2, \ldots, v_n)$ em F tal que, para cada mapeamento μ em V, temos $(sum(v, v_1, v_2, \ldots, v_n))\,\mu = \mu'$, onde $\mu'(v) = c_1 + c_2 + \cdots + c_n$ se $\mu(v_i) = c_i$ para todo i entre 1 e n. Se pelo menos um dentre $\mu(v_1), \ldots, \mu(v_n)$ for *não-constante*, e nenhum for *indefinido*, então $\mu'(v) = \text{não-constante}$. Se um dentre $\mu(v_1), \ldots, \mu(v_n)$ for *indefinido*, mas nenhum deles é *não-constante*, o resultado é a função indefinido. Isto, função expressa o efeito da atribuição $v := v_1 + v_2 + \cdots + v_n$. Como usual neste capítulo, $+$ pode ser pensado como um operador genérico, apesar de uma ou mais vezes é necessário se o operador for unário ou binário ou algum outro número de operandos e uma ou mais vezes modificação é necessária para levar em conta o efeito de um enunciado de atribuição $\ldots$
4. Para cada variável v existe uma função $kill(v)$ em F tal que, para cada μ, $(kill(v))\,\mu = \mu'$, onde $\mu'(v) = \ldots$ para todo $\ldots$ que não $\ldots$

	não-constante	c	d, $c \neq d$	*indefinido*
não-constante	*não-constante*	*não-constante*	*não-constante*	*não-constante*
c	*não-constante*	c	*não-constante*	c
indefinido	*não-constante*	c	d	*indefinido*

Fig. 10.58. $\mu_1 \sqcap \mu_2 = \mu$ em termos de $\mu_1(v)$ e de $\mu_2(v)$.

não convergem. Esta função reflete uma definição através de uma lacuna: de x_0 para cima após a primeira de leitura, x certamente precisa ser presumido como um conhecido qualquer valor constante particular. □

Os Axiomas das Estruturas da Análise de Fluxo de Dados

A fim de fazer funcionar, para uma estrutura, algoritmos de análise de fluxo de dados que vimos até aqui, precisamos fazer algumas suposições a respeito do conjunto L do conjunto de funções F e do operador de reunião $\wedge$. Nossas suposições básicas são listadas abaixo, apesar de alguns algoritmos poderem fazer uso de suposições adicionais.

1. F possui uma função identidade id, tal que $id(x) = x$ para todo x em L.

2. F é fechado sob composição de funções; isto é, para quaisquer duas funções f e g em F, a função h definida por $h(x) = g(f(x))$ está em F.

3. $\wedge$ é uma operação associativa, comutativa e idempotente. Essas três propriedades são expressas algebricamente como

$$\begin{aligned} x \wedge (y \wedge z) &= (x \wedge y) \wedge z \\ x \wedge y &= y \wedge x \\ x \wedge x &= x \end{aligned}$$

para todos os x, y e z em L.

4. Existe um elemento de *topo* $\top$ que satisfaz a lei

$$\top \wedge x = x$$

para todo x em L.

Exemplo 10.43. Vamos considerar as definições incidentes I certamente possuem a identidade, a função onde os conjuntos *gen* e *kill* são ambos o conjunto vazio. Para mostrar o fechamento sobre a composição de funções, suponhamos herdadas funções

$$\begin{aligned} f_1(X) &= G_1 \cup (X - K_1) \\ f_2(X) &= G_2 \cup (X - K_2) \end{aligned}$$

Então,

$$f_2(f_1(X)) = G_2 \cup (G_1 \cup (X - K_1) - K_2)$$

Podemos verificar que o lado direito da igualdade acima é algebricamente igual a

$$(G_2 \cup (G_1 - K_2)) \cup (X - (K_1 \cup K_2))$$

Se fizermos $G = G_2 \cup (G_1 - K_2)$ e $K = K_1 \cup K_2$, constatamos, então, que a composição de f_1 e f_2, a qual é $f(X) = G \cup (X - K)$, é da forma que a torna um membro de F.

Para o operador de reunião, que é também associativo, verificar que a união é associativa, comutativa e idempotente. O elemento de "topo" será o conjunto vazio neste caso, uma vez que $\emptyset \cup X = X$ para todo conjunto X.

Quando considerarmos as expressões disponíveis, encontramos que nossos argumentos usados para as definições incidentes também mostrarão que F possui uma identidade e é fechado sobre composição de funções. O operador de convergência é agora a interseção, mas esse operador é também associativo, comutativo e idempotente. O elemento de topo faz mais sentido do ponto de vista intuitivo desta vez: é o conjunto U de todas as expressões no programa, porque para qualquer conjunto X de expressões, $U \cap X = X$. □

Exemplo 10.44. Vamos considerar a estrutura de conjunto de constantes inovadoras dos Exemplos 10.12. O conjunto de funções F foi projetado para incluir a identidade e ser fechado sob composição de funções. Para verificar as leis algébricas de $\wedge$, basta mostrar que as mesmas se aplicam a cada variável v. Como uma instância, vamos verificar a idempotência; isto é, para todo x, $x \wedge x = x$. Se $x(v) = \top$, $\bot$ ou c, é simples verificar os casos em que $x \wedge x = x$. Por exemplo, se $x(v) = \text{não-constante}$, então $x \wedge x = \text{não-constante}$, pois a regra para o cruzamento de *não-constante* com qualquer coisa é *não-constante*, na Fig. 10.38. *É assim em todos*.

Finalmente, o elemento de topo é simplesmente o elemento x tal que $x(v) = \text{indefinido}$ para todas as variáveis v. Podemos verificar a partir da Fig. 10.38, que para qualquer mapeamento x e qualquer variável v, se a função $f = x \wedge y$, então $f(v) = x(v)$, já que o resultado é exatamente de *não definido* com qualquer valor, na Fig. 10.38, é aquele outro valor. □

Monotonicidade e Distributividade

Precisamos de outra condição para fazer os algoritmos iterativos para a análise de fluxo de dados funcionar. Esta condição, chamada de monotonicidade, diz informalmente que se tomarmos uma função qualquer g conveniente de conjuntos e aplicarmos a dois membros de L, um "maior" do que o outro, o resultado da aplicação de g a um não é menor do que o que obtemos aplicando g ao menor.

Para formar a noção de "maior" precisamos definir uma relação $\leq$ em L por

$$x \leq y \text{ se e somente se } x \wedge y = x$$

Exemplo 10.45. No exemplo das definições incidentes, onde a reunião é união e os membros de L são conjuntos de definições, $X \leq Y$ significa $X \cup Y = Y$, isto é, X é um subconjunto de Y. Por conseguinte, $\leq$ parece "pequeno" os membros elementos de L são superconjuntos dos menores.

Para as expressões disponíveis, onde a convergência é interseção, $\leq$ é o oposto: "grande" significa que $X \cap Y = X$, isto é, X é um subconjunto de Y. □

Notemos, a partir do Exemplo 10.45, que $\leq$ é uma ordem significa não precisa ter todas as propriedades de $\leq$ sobre os inteiros. É verdadeiro que $\leq$ é transitiva, e também pode provar, como um exercício, que os dois axiomas de $\wedge$ que $x \leq y$ e $x \leq y$ implicam que $x = y$. No entanto, em nossos semilattices não é uma relação de ordem total. Por exemplo, em nosso conjunto de expressões disponíveis, podemos ter dois conjuntos $X \neq Y$ nenhum dos quais seja um subconjunto do outro, caso em que nem $X \leq Y$ nem $Y \leq X$ é verdadeiro.

Frequentemente ajuda descobrir o conjunto V cujo diagrama de Hasse é idêntico ao qual é um grafo cujos nós são os elementos de L e cujos arcos são dirigidos para baixo de X para Y se $X > Y$. Por exemplo, a Fig. 10.50 mostra o conjunto V para o problema de fluxo de dados das definições incidentes, onde existem três definições d_1, d_2 e d_3. Lembre que V é um "superconjunto de", conjuntos dirigido para baixo a partir de qualquer subconjunto dessas três definições e os conjuntos superconjuntos. Já que $\leq$ é transitiva, evidentemente continuamos o caminho para baixo de X para Y se existir um outro percurso de X para Y desenhado no diagrama. Por conseguinte, apesar de $\{d_1, d_2\} \leq \{d_1\}$, não desenhamos este lado, uma vez que já está representado pelo percurso, através de $\{d_1, d_2\}$, por exemplo.

É também útil notar que podemos extrair as operações de reunião a partir do diagrama, já que $X \wedge Y$ é sempre o maior Z para o qual existem percursos para baixo até Z a partir de ambos X e Y. Por exemplo, se $X = \{d_1, d_2\}$ e $Y = \{d_1, d_3\}$, então Z na Fig. 10.50 é $\{d_1\}$, o que faz sentido, porque o operador de reunião é união. É também verdadeiro que o elemento de topo $\top$ aparece abaixo de qualquer

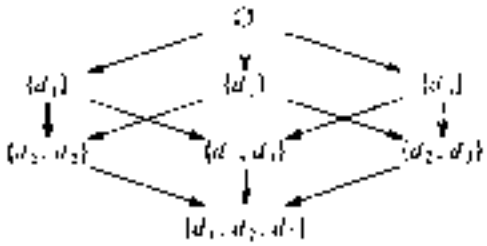


Fig. 10.59. Reticulado de subconjuntos de dois elementos.

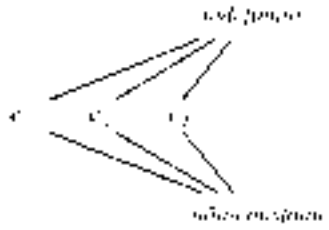


Fig. 10.60. Diagrama reticulado para valores de verdade.

maximalidade, isto é, existe um percurso para baixo de 1 para cada elemento.

Podemos agora definir uma estrutura $(V, \wedge, \vee, \neg, 0, 1)$ como anteriormente:

$$p \wedge q = r \text{ implica que } p(x) \wedge q(x) = r(x) \tag{10.51}$$

para todo p, q, r em V e x em U.

Existe uma forma equivalente de se definir a mesma comutatividade

$$p(x) \wedge q(x) = q(x) \wedge p(x) \tag{10.52}$$

para todo p e q em V e x em U. É útil se ter em vista essas duas definições equivalentes, de forma que iremos definir uma prova de uma equivalente dessa, pois para definir a conjunção de algumas proposições simples, usamos a definição de $\wedge$ e não essencialmente somente $\wedge$ de conjunção para

Vamos assumir (10.51) e provar por que (10.16) é válida. Notemos, primeiro, que, para se obter $p \wedge q = r$, p e q são ambos valores de x. Uma prova simples de (10.16) para o caso em que $p(x) = q(x)$ é dada por (10.51). Vejamos agora o caso em que $p(x) \neq q(x)$. Como $p \wedge q = r$, $p(x) \wedge q(x) = r(x)$. Mas, de acordo com a definição de $\wedge$, $p(x) \wedge q(x) = p(x)$ ou $q(x)$. Mas, de acordo com a definição de $\wedge$, $p(x) \wedge q(x) = p(x)$ ou $q(x)$. Como $p(x) \neq q(x)$, $r(x) = p(x)$ ou $r(x) = q(x)$. Como uma negação geral, temos que $r(x) = p(x)$ ou $r(x) = q(x)$.

$$r(x) = p(x) \text{ ou } r(x) = q(x).$$

A consequência acima (10.16) implica que $p \wedge q = r$, e, assim, provamos (10.51).

Precisamente, uma estrutura obedece a uma condição mais forte que (10.16), que chamamos de *condição de distributividade*:

$$p(x) \wedge (q(x) \vee r(x)) = (p(x) \wedge q(x)) \vee (p(x) \wedge r(x))$$

para todo p, q, r em V e x em U. Certamente se $p = q$, então $p \wedge q = p$ pela definição de $\wedge$. Assim, a distributividade implica a comutatividade.

Exemplo 10.46. Consideremos as estruturas de definições incidentes. Seja $V = \{0, 1/2, 1\}$ o conjunto de definições e seja f uma função definida por $f(x) = 1 - x$, $0 \leq x \leq 1$, para a mais comum de definições $[x, M]$. Podemos provar, então, que a estrutura das definições incidentes satisfaz a condição de distributividade, através da verificação de que

$$p \wedge (q \vee r) = (p \wedge q) \vee (p \wedge r)$$

Ou descobrir de um diagrama de Venn. Uma transparência a prova de que a comutatividade para operações de conjunção pode ser complicada.

Exemplo 10.47. Vamos mostrar, agora, que existem duas definições incidentes que não satisfazem a condição de distributividade. Primeiro, definimos a operação $\wedge$ de conjunção através da tabela da Fig. 10.58. Isto é, vamos definir

$$\text{valor verdade } p \wedge q = \text{mínimo valor verdade para qualquer constante } x$$

$$p \wedge q = \begin{cases} r, & \text{se } p(x) = q(x) = r \text{ para qualquer constante } x \\ \text{valor falso}, & \text{se } p(x) = \text{valor falso para qualquer constante } x \\ \text{indefinido}, & \text{se } p(x) = \text{valor verdade e } q(x) = \text{indefinido} \\ & \text{ou } p(x) = \text{indefinido e } q(x) = \text{valor verdade} \end{cases}$$

A Fig. 10.58 pode ser interpretada como dizendo que $p \wedge q = r$ se $p(x) = q(x) = r$ para qualquer x.

Podemos determinar que a relacionamento $\leq$ é obtido a partir da operação $\wedge$. Encontramos que

$$\begin{aligned} &\text{valor verdade } p \leq q \text{ se } p \wedge q = p \text{ para qualquer constante } x \\ &\text{valor falso para qualquer constante } x \\ &\text{valor verdade} = \text{indefinido} \end{aligned}$$

Este relacionamento é mostrado no diagrama reticulado da Fig. 10.60, onde os vértices destacados são os elementos de V que possuem valores verdade. Notemos que aqui a ligação entre dois elementos de V que possuem uma relação sob esses conjuntos de valores para x e para valores indefinidos de x. Os conjuntos de U podem ser pensados como valores de uma variável, um componente para cada variável em um diagrama reticulado. Cada ponto representado na parte de baixo da Fig. 10.60 se relaciona com os pontos representados por cima da linha, com cada componente relacionado por uma relação $\leq$ com o componente correspondente.

Por conseguinte, dizer que $p \leq q$ é dizer que sempre que p for uma constante x, q também é constante em x, ou seja, que $p \leq q$ se, e somente se, $p \wedge q = p$. Uma consequência desta definição é que a verdade para $p \leq q$ implica $p \wedge q = p$ para todo p e q. Por exemplo, $p \leq q$ se p é indefinido e q é verdade, mas $p \wedge q = p$ se p é indefinido e q é verdade. Precisamos constatar todas as possíveis valores de p e q para ver se $p \leq q$ implica $p \wedge q = p$. Estudando as restrições sobre p e q para que $p \leq q$ e $p \wedge q = p$, obtemos o exemplo:

$$\begin{aligned} &p(x) = \text{valor verdade} \\ &q(x) = 2 \\ &p(x) = 1 \\ &q(x) = \text{valor falso} \end{aligned}$$

Isto é, $p \leq q$ se p é verdade e q é falso, mas $p \wedge q = p$ se p é verdade e q é falso. Isto é, $p \leq q$ se p é verdade e q é falso, mas $p \wedge q = p$ se p é verdade e q é falso.

[illegible] = não constante e [illegible] = não constante. Uma vez que não constante [illegible] cada função [illegible] a verificação em um caso. Os outros casos são deixados para o leitor como exercício.

Precisamos agora verificar nossa afirmação de que a estrutura de cômputo de constantes não é distributiva. Para esta parte seja a função associada à atribuição [illegible] = 3 [illegible] = 2. Seja [illegible]. Então [illegible] = 2 [illegible] = não constante. Similarmente [illegible] = não constante. Equivalentemente [illegible] = não constante. Segue-se que [illegible] = não constante, uma vez que a soma de dois valores não constantes é, presumidamente, não constante.

Por outro lado, [illegible] = 5, já que dado [illegible] = 2 e [illegible] = 3, a atribuição [illegible] faz [illegible] igual a 5. Similarmente, [illegible] = 5. Por conseguinte, [illegible] = 5. Vemos agora que [illegible] = [illegible] e dessa forma a distributividade é violada.

Intuitivamente, a razão pela qual chegamos a uma violação da distributividade está em que a estrutura de cômputo de constantes não é poderosa o suficiente para se lembrar de todas as invariantes e, em particular, o fato de que ao longo de percursos cujos efeitos sobre as variáveis são descritos [illegible] a mesma [illegible] estará em vigor ainda que nenhuma [illegible] sejam pressa mesmas constantes. Podemos usar outras estruturas mais complicadas para evitar esse problema particular, apesar de [illegible] Felizmente, a distributividade é aceitável para um algoritmo de fluxo de dados iterativo "funcionar", como veremos em seguida.

Soluções de Convergência de Percursos para Problemas de Fluxo de Dados

Vamos imaginar que um grafo de fluxo tenha associado a cada um de seus nós uma função de transferência, uma das funções no conjunto F. Para cada bloco B, seja f_B a função de transferência para B.

Consideremos qualquer percurso P: $B_0 \rightarrow B_1 \rightarrow \ldots \rightarrow B_k$ a partir do nó inicial B_0 até algum bloco B_k. Podemos definir a *função de transferência para* P como sendo a composição de $f_{B_0}, f_{B_1}, \ldots, f_{B_{k-1}}$. Note que f_{B_k} não é parte da composição, refletindo o ponto de vista de que este percurso é tomado para atingir o início do bloco B_k, não o seu fim.

Assumimos que os valores em V representam informações a respeito de dados usados pelo programa e que o operador de confluência $\wedge$ nos diz como aquelas informações são combinadas quando os percursos convergem. Também faz sentido ver o elemento de topo como representando "nenhuma informação", uma vez que qualquer percurso que carregue o elemento de topo pode desaguar em qualquer outro percurso, dependendo de que informações serão combinadas após a confluência dos percursos. Por conseguinte, se B é um bloco no grafo de fluxo, as informações que entram em B deveriam ser computadas através da consideração de cada possível percurso a partir do nó inicial até B e vendo o que acontece ao longo de cada um desses percursos, começando-se sem nenhuma informação. Isto é, para cada percurso P de B_0 até B computamos $f_P(\top)$ e tomamos a convergência de todos os elementos resultantes.

Em princípio, essa convergência poderia ser sobre um número infinito de diferentes valores, uma vez que existe um número infinito de diferentes percursos. Na prática, é frequentemente aceitável considerar somente os percursos acíclicos e, mesmo quando não seja, como para a estrutura de cômputo de constantes, discutida acima, existem usualmente outras razões que podemos encontrar para tornar finita essa convergência infinita, para qualquer grafo de fluxo particular.

Formalmente, definimos a *solução de convergência de percursos* para um grafo de fluxo como sendo

$$mop(B) = \bigwedge_{\substack{\text{percursos } P \\ \text{de } B_0 \text{ a } B}} f_P(\top)$$

A solução *mop* para um grafo de fluxo faz sentido quando apreendemos que, no que diz respeito às informações que atingem o bloco B, são consideradas o grafo de fluxo pode ser aquele sugerido na Fig. 10.61, onde a cada função de transferência associada a cada um dos percursos $P_1, P_2, \ldots$ (um número de percursos possivelmente infinito) [illegible] de fluxo original, [illegible] percurso inteiramente distinto até B. Na Fig. 10.61, as informações que atingem B são dadas pela convergência de todos os percursos.

Soluções Conservativas para Problemas de Fluxo de Dados

Quando tentamos solucionar equações de fluxo de dados que vêm de uma estrutura arbitrária, podemos estar em tais equações não apresenta a solução *mop* facilmente. Felizmente, da mesma forma que com os exemplos concretos das estruturas de fluxo de dados nas Seções 10.5 e 10.6, existe uma direção segura na qual erramos e o algoritmo de fluxo de dados iterativo [illegible] naquelas seções se presta sempre a uma [illegible] solução segura. Dizemos que uma solução $in[B]$ é uma solução segura se $in[B] \leq mop[B]$ para todos os blocos B.

A despeito de [illegible] definição da "estratégia" [illegible] Lembremos que em qualquer grafo de fluxo, o conjunto de percursos aparentes até um nó aqueles que sejam percursos no grafo de fluxo pode ser um superconjunto próprio dos percursos reais, aqueles que são seguidos em alguma execução do programa correspondente ao grafo de fluxo em questão. Para que o resultado da análise de fluxo de dados seja usável, seja qual for o propósito, os dados precisam estar seguros se modificarmos o grafo de fluxo removendo alguns dos percursos, uma vez que não podemos, em geral, distinguir os percursos reais dos percursos aparentes que não são reais.

Suponhamos que dentre o conjunto infinito de percursos sugerido pela Fig. 10.61, seja a convergência [illegible] todos os percursos reais P que sejam seguidos em alguma execução. [illegible] seja a convergência de [illegible] sobre todos os outros percursos P. Por conseguinte, $mop(B) = x \wedge y$. Então, a resposta verdadeira para nosso problema de fluxo de dados é [illegible], mas a solução *mop* é [illegible]. Relembremos que [illegible], uma vez que [illegible]. Por conseguinte, a solução *mop* [illegible] solução verdadeira.

Enquanto possamos preferir a solução verdadeira para o problema de fluxo de dados, quase certamente não teremos forma eficiente de dizer exatamente quais percursos são reais e quais não são, de forma que somos forçados a aceitar a solução *mop* como a solução segura mais próxima. Por conseguinte, qualquer que seja o uso que façamos das informações de fluxo de dados, é necessário que seja consistente com a possibilidade da solução que obtemos ser $\leq$ que a solução verdadeira. Uma vez que aceitamos isso, devemos também ser capazes de aceitar uma solução que seja $\leq$ que a solução *mop*, por conseguinte, $\leq$ a solução verdadeira. Tais soluções são mais fáceis de se obter do que a *mop* para aquelas estruturas que são monótonas mas não distributivas. Para estruturas distributivas como aquelas da Seção 10.6, o algoritmo iterativo simples computa a solução *mop*.

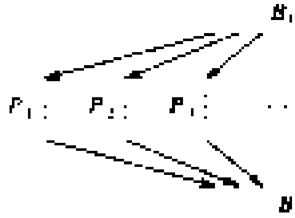


Fig. 10.61. Grafo mostrando o conjunto de todos os percursos possíveis para B.

[illegible]

```
(11)  para cada nó B faça        inicializar, assumindo-se que entrada[B] = ⊤
(2)       saída[B] = g_B(⊤)
(3)  enquanto houver alterações em qualquer saída faça
(4)       para cada bloco B, em ordem de pesquisa em profundidade faça início
(5)            entrada[B] = ⊓_{P ∈ pred(B)} saída[P]
                   { ação a ser tomada em um ponto de junção }
(6)            saída[B] := g_B(entrada[B])
               fim
```

Fig. 10.62 Algoritmo iterativo para estruturas gerais.

O Algoritmo Iterativo para Estruturas Gerais

Existe uma generalização óbvia do Algoritmo 10.2 que funciona para uma ampla variedade de estruturas. O algoritmo deixa claro o respeito que a estrutura especial do fluxo de controle tem sobre a convergência sobre o conjunto mínimo de pontos fixos, sugerida na Fig. 10.61, e especifica a convergência sobre um subconjunto finito. Examinaremos o algoritmo e em seguida discutiremos as formas pelas quais podemos assegurar a finitude. Nem sempre uma garantia constante de finitude é aquela que nos acompanhará o tempo todo: a propagação só longe de pontos fixos mínimos é suficiente.

Algoritmo 10.18. Solução iterativa para estruturas gerais de fluxo de dados.

Entrada. Um grafo de fluxo de dados, no qual cada nó *B* tem uma função de transferência g_B, um conjunto de ninguéns $\sqcap$, uma operação de reunião $\sqcap$ e uma atribuição de um valor de V a cada nó do grafo de fluxo.

Saída. Um valor *entrada*[*B*] em V para cada nó do grafo de fluxo.

Método. O algoritmo é dado pela Fig. 10.62. Diferentes interpretações dos algoritmos iterativos de fluxo de dados já familiares, computamos *entrada* e *saída* para cada um deles de aproximações sucessivas. Assumimos que g_B é a função em *V* associada ao bloco *B*; aquela função desempenha o papel de *gerados* e *mortos* na Seção 10.1.

Uma Ferramenta de Análise de Fluxo de Dados

Podemos ver agora como os vários dados vistos até aqui podem ser aplicados a uma ferramenta para análise de fluxo de dados. O Algoritmo 10.18 depende, para seu funcionamento, de três suposições básicas:

1. Uma regra, por exemplo f_B, por *B* e um dado cabal em *V*. Esta regra é usada nas linhas (2) e (6) da Fig. 10.62.
2. Uma regra para aplicar o operador de reunião a dois valores em *V*, esta regra é usada na linha (5) e também nas linhas (1) e (2).
3. Uma regra para informar se dois valores são iguais. Este teste não é feito explicitamente na Fig. 10.62, mas é implícito no teste para a mudança em qualquer dos valores de *saída*.

Nessas estruturas, também mencionadas nos dados vistos até aqui, para *V* e $\sqcap$ de forma que podemos transpor argumentos as objeções mencionadas acima. Os valores de *entrada* e *saída* na Fig. 10.62, independentemente do tipo dos dados para *V*. Um fluxo, nesses termos, é uma forma que irá formar a representação conhecida do conteúdo de cada bloco básico, isto é, uma lista de comandos e produzir um elemento de *V* que seja função de transferência para aquele bloco.

Exemplo 10.48. Para estruturas de definições reaching, podemos construir uma tabela que identifica associada a cada bloco do grafo de fluxo dado com um único número de 1 até algum máximo *n* pode ser representado por partes de vetores de bits daquele tamanho. Isto é, pelos conjuntos gerados e mortos. A ideia por trás destas vetores de bits para *entrada* e *saída*, dados os estados de um bloco e uma tabela que associa cada um dos definições com posições no vetor de bits, é direta, com as operações usuais para conjuntos relativos com lógica de vetores de bits. Podemos representar qualquer de vetores de bits e aplicar funções definidas por um programa usando vetores de bits.

A heterogeneidade, de análise de fluxo de dados e, portanto, um projeto muito diferente de uma implementação da Fig. 10.62 como biblioteca para as subrotinas dadas se aplica que [illegible] necessitaria uma escolha aplicação de função ou computação. A ferramenta suportaria uma representação lista de grafos de fluxo e, portanto, estaria capacitada a realizar tarefas como encontrar todos os predecessores de um nó, encontrar o ordenamento em profundidade de um grafo de fluxo e aplicar a cada bloco a regra que compõe a função em *V* associada aquele bloco. A vantagem de se usar uma tal ferramenta está em que a manipulação de grafos e os aspectos da teoria, que dão convergência do Algoritmo 10.18 também que ser escritos para cada análise de fluxo de dados que fazemos.

Propriedades do Algoritmo 10.18

Devemos agora declarar as suposições sobre as quais o Algoritmo 10.18 é eventualmente funciona e exatamente para que o algoritmo converge, quando realmente converge. Primeiro, é uma estrutura finita com vetores afirmamos então, que o resultado para o algoritmo é que *entrada*[*B*] $= \sqcap_P$ *saída*[*P*] para todos os blocos *B*. A razão intuitiva é que o longo de qualquer percurso $P = B_1 B_2 \ldots B_k$ a partir do nó inicial até $B = B_k$, podemos mostrar por indução sobre *i* que o efeito do percurso a partir de B_1 até B_i é sempre após o máximo de *i* iterações do laço **enquanto** da Fig. 10.62. Isto é, se *P* é o percurso $B_1 \ldots B_i$, então, após *i* rodadas, *entrada*[*B*] $= f_P(\top)$. Por conseguinte, qualquer vez o algoritmo converge, *entrada*[*B*] até $\sqcap_P f_P(\top)$ para cada percurso *P* de B_1 até *B*. Usando a regra que estabelece que se $x \sqcap y = x$, então $x \sqsubseteq y$, podemos mostrar que *entrada*[*B*] $\sqsubseteq f_P(\top)$.

Quando uma estrutura é distributiva, podemos mostrar que o Algoritmo 10.18 converge, de fato, para a solução ideal. A função essencial é provar que todas as vezes durante a execução do algoritmo, *entrada*[*B*] e *saída*[*B*] são cada um, iguais a reunião de $f_P(\top)$ para algum conjunto de percursos *P* até a única final de *B*, respectivamente. Veremos, no entanto, no próximo exemplo, que este não precisa ser o caso quando a estrutura [illegible] é distributiva.

[illegible]

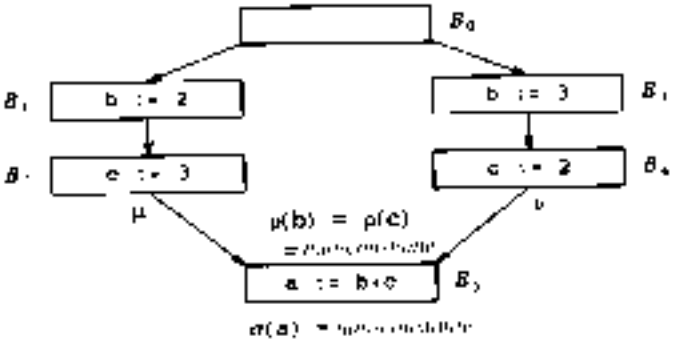


Fig. 10.63. [illegible]

Exemplo 10.49. Vamos explanar o exemplo com mais detalhes. Lembrando da estrutura de computação de constantes discutida no Exemplo 10.17, o grafo de fluxo relevante é mostrado na Fig. 10.63. Os mapeamentos ρ e σ variando de B_0 a B_4 são aqueles mostrados no Exemplo 10.47. O mapeamento associado à entrada B_5 é ρ [illegible] Lembre-se também que cada percurso real de cada entrada apresenta computação $\sigma = S$ após B.

O problema, mais uma vez, é com o Algoritmo 10.18. Usando uma estrutura mais detalhada, se computa como se algumas seqüências de nós, que não são de fato possíveis percursos, também operassem sobre grafo de fluxo. Assim, por exemplo, na Fig. 10.63, o algoritmo se computa como se os percursos como $B_1 \to B_2 \to B_3 \to B_5$ ou $B_1 \to B_2 \to B_3 \to B_4 \to B_5$ fossem percursos reais estabelecendo c com uma combinação de valores que não seria [illegible]

Convergência do Algoritmo 10.18

Existem várias formas pelas quais poderíamos provar que o Algoritmo 10.18 converge para uma solução particular. Provavelmente, como somamos contribuições de percursos cíclicos sejam necessários, isto é, poderíamos mostrar que a iteração sobre percursos cíclicos é monótona e que a solução cai sobre todos os percursos, se existir um, mas somente depois que o algoritmo convergir. Mais formalmente, seja ρ uma função de valores de entrada convergente, ρ' uma função de valores de entrada após uma passagem a mais do que a profundidade do grafo de fluxo, como discutimos na Seção 10.10.

Por outro lado, estruturas como nosso exemplo de computação de constantes, requerem a consideração de mais de quatro percursos acíclicos. Por exemplo, a Fig. 10.64 mostra um simples grafo de fluxo onde temos que considerar o percurso $B_1 \to B_2 \to B_3 \to B_2 \to B_3$, para concluirmos que x não tem um valor constante ao entrar em B_4.

No entanto, para computações constantes podemos argumentar que o Algoritmo 10.18 converge como se segue. Primeiro, é fácil ver que, para uma estrutura monótona arbitrária que *entrada*[B] e *saída*[B], para todo bloco B, tem-se uma sequência não crescente, mas decrescente de que o novo valor para uma dessas variáveis é sempre $\leq$ que o valor antigo. Se relembrarmos a Fig. 10.10, o diagrama reticulado para os valores de nó, para nós aplicarmos as variáveis computacionais que

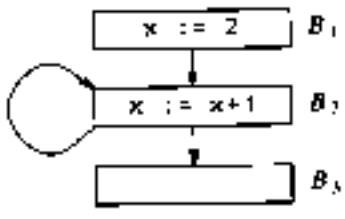


Fig. 10.64. [illegible]

para qualquer variável o valor de *entrada*[B] ou de *saída*[B] pode cair apenas duas vezes, uma após ser definido pela primeira constante e outra daquela constante para *não-constante*.

Suponhamos que existam q nós e v variáveis. Então, a cada iteração do laço **enquanto** da Fig. 9.62, pelo menos uma variável passa a ter a qualidade de seu valor em algum conjunto *saída*[B] ou algum conjunto *entrada*[B] descer. Assim, o algoritmo converge no mesmo mais iteração do que a soma do número de mudanças dos valores dos conjuntos citados acima. Portanto, o número de iterações está limitado a $2qv$, ou, como tal número de mudanças é também a complexidade, tem que ter ao menos o valor máximo [illegible] a cada bloco do grafo de fluxo.

Ajustando a Inicialização

Em alguns problemas de fluxo de dados existe uma dependência entre o que o Algoritmo 10.18 nos dá como solução e o que desejamos realmente. Relembre-se que para as expressões disponíveis [illegible] a interseção de todos os conjuntos de todas as expressões. Uma vez que o Algoritmo 10.18 assume implicitamente que *entrada*[B] $= \emptyset$ para cada bloco B, incluindo-se no final a solução [illegible] pelo Algoritmo 10.18 é efetivamente o conjunto de expressões que, assumindo que as mesmas estejam disponíveis no início [illegible] estão disponíveis na entrada do bloco B.

A diferença, naturalmente, é que poderíamos existir percursos a partir do início até B nos quais uma expressão $x + y$ não é sempre gerada nem morta. O Algoritmo 10.19 diria que $x + y$ está disponível quando de fato não está, porque nenhuma variável ao longo daquele percurso pode ser alterada guardando nosso valor. A correção é simples. Podemos ou modificar o Algoritmo 10.18 de forma que para a estrutura de expressões disponíveis, *entrada*[B_1] seja estabelecido igual ao conjunto vazio, ou podemos modificar o grafo de fluxo introduzindo um nó inicial fictício, com predecessor do início real, que mata cada expressão.

10.12 ESTIMATIVA DE TIPOS

Chegamos agora a um problema de flow-graph dos que é mais desafiador do que aqueles estudados na seção anterior. Várias linguagens, tais como APL e SETL, nos quais objetos de tipo não especificado são os tipos das variáveis sejam declarados e permitem, inclusive, que a mesma variável abrigue valores de diferentes tipos em diferentes tempos. Tentativas sérias de compilar tais linguagens em código eficiente têm usado a análise de fluxo de dados para inferir os tipos de variáveis, já que, naturalmente, o código gerado para adicionar dois objetos dentro de uma variável de tipos possíveis (por exemplo, inteiros, reais, vetores).

Nossa primeira estimativa seria a de que o cômputo dos tipos das variáveis seria algo como computar as definições incidentes. Podemos associar um conjunto de possíveis tipos para cada variável a cada ponto. O operador de confluência é a união sobre os conjuntos de tipos, já que se a variável x possui o conjunto de tipos possíveis S em um percurso e o conjunto T em outro, então x pode possuir qualquer um dos tipos em $S \cup T$, após a confluência dos percursos. Na entrada, com que o percurso passe através de um ponto onde devemos estar capazes de realizar algumas inferências a respeito dos tipos das variáveis baseados nos operadores dos quais nós possuímos [illegible] operandos e nos tipos que produzem como resultado. Um exemplo é o que lidou com um operador que podia multiplicar dois números inteiros quando complexos. Há um exemplo desse tipo de inferência.

Infelizmente, existem pelo menos dois problemas com essa abordagem.

1. Os conjuntos de possíveis tipos para uma variável pode ser muito
2. A determinação de tipos muitas vezes requer uma propagação de informações para adiante quanto para trás. Em [illegible] estimativas

precisar dos possíveis tipos. Por conseguinte, mesmo a estrutura da Seção [illegible] é [illegible] suficiente para fazer justiça ao problema.

Antes de começar a parte [illegible] vamos examinar alguns tipos de inferências a respeito de tipos que podem ser feitas em algumas linguagens típicas.

Exemplo 10.50. Consideremos os enunciados

$$[illegible]$$

Suponhamos que, em primeiro lugar, não saibamos nada a respeito dos tipos das variáveis [illegible]. No entanto, vamos admitir que o operador de acesso a array [illegible] requeira um argumento inteiro. Pelo exame do primeiro enunciado, podemos inferir que [illegible] seja um inteiro e aquele [illegible] seja um array de elementos de algum tipo. Do segundo, [illegible] [illegible].

Agora, devemos propagar as inferências para trás. Se [illegible] for computado como sendo um inteiro, no primeiro enunciado, então o [illegible] expressão [illegible] precisa ser inteiro, o que significa que [illegible] precisa ser um array de inteiros. Podemos, então, raciocinar para frente [illegible] que o valor atribuído a [illegible] pelo segundo enunciado precisa também ser um inteiro. Note-se que é impossível descobrir que os elementos de [illegible] são inteiros raciocinando-se somente para a frente ou para trás somente.

Lidando com Conjuntos Infinitos de Tipos

Existem numerosos exemplos de casos para os quais o conjunto de tipos possíveis para uma variável é realmente infinito. Por exemplo, SETL permite que um enunciado como

$$[illegible]$$

seja executado dentro de um laço. Se começarmos sabendo somente que [illegible] poderia ser um inteiro, após essas iterações uma iteração do laço, compreenderemos que [illegible] poderia ser um inteiro ou um conjunto de inteiros. Após considerar uma segunda iteração, encontramos que [illegible] poderia ser um conjunto de um conjunto de inteiros, e assim por diante.

Um problema similar poderia ocorrer numa versão sem tipos de uma linguagem convencional, como C, onde o enunciado

$$[illegible]$$

com a possibilidade inicial de que [illegible] seja um inteiro, nos levaria a descobrir que [illegible] pode ter qualquer tipo da forma

apontador para apontador para ... apontador para inteiro

A forma tradicional para se lidar com tais problemas é reduzir o conjunto de possíveis tipos a um número finito. A idéia geral é a de agrupar o número infinito de possíveis tipos num número finito de classes, geralmente mantendo-se os tipos mais simples isolados e agrupando os mais complicados [illegible] em classes amplas. Quando assim o fazemos, temos de exercitar o julgamento sobre como fazer as inferências a respeito das interações entre os tipos e os operadores. O seguinte exemplo sugere o que pode ser feito.

Exemplo 10.51. Vamos continuar o exemplo do Capítulo 6, onde usamos o operador [illegible] para construir tipos para funções. Aqui, nosso conjunto de tipos inclui o tipo básico *int* e todos os tipos da forma [illegible], representando o tipo de uma função com domínio de tipo [illegible] e intervalo de tipo [illegible], onde [illegible] são tipos no nosso conjunto. Por conseguinte, o conjunto de tipos é infinito, incluindo tipos tais como

$$[illegible]$$

Para reduzir este conjunto a um número finito de classes, iremos restringir uma expressão de tipo, de forma que tenha somente um construtor do tipo função [illegible], substituindo subexpressões numa expressão de tipo contendo pelo menos um construtor de tipos por *func*. Por conseguinte, existem cinco diferentes tipos:

int
int → *int*
int → *func*
func → *int*
func → *func*

Iremos representar os conjuntos de tipos como vetores de bits de comprimento cinco, com as posições correspondendo aos cinco tipos listados na ordem acima. Por conseguinte, 01111 representa o tipo para qualquer aplicação de função, isto é, para qualquer coisa que não um *int*. Note-se que este é, num certo sentido, o tipo de *func*, uma vez que não pode ser um inteiro.

O enunciado de atribuição básico para o nosso modelo é

$$[illegible]$$

Sabendo os possíveis tipos de [illegible] e de [illegible], podemos determinar os possíveis tipos de [illegible], procurando-os na tabela da Fig. 10.65. Se [illegible] puder ser qualquer tipo no conjunto S_1 e [illegible] qualquer tipo no conjunto S_2, tomamos cada par τ em S_1 e σ em S_2, e procuramos pela entrada na linha para τ e coluna para σ, o que chamamos $\tau m \sigma$. Em seguida, tomamos a união dos resultados de todas essas buscas para obter o conjunto de possíveis tipos para [illegible].

Por exemplo, se $\tau = int \rightarrow func$ e $\sigma = int$, então, $\tau m \sigma = 01111$. Isto é, o resultado de se aplicar um mapeamento do tipo *int* → *func* a um *int* é uma *func*, o que significa um mapeamento de qualquer um dos quatro tipos além de *int*. Não podemos dizer qual porque nossa simplificação de um número infinito de tipos a apenas cinco classes nos impede de saber.

Para um segundo exemplo, seja τ como antes e $\sigma = int \rightarrow int$. Então, $\tau m \sigma = 00000$, porque o tipo do domínio de τ é definitivamente

τ	σ				
	int	*int* → *int*	*int* → *func*	*func* → *int*	*func* → *func*
int	00000	00000	00000	00000	00000
int → *int*	10000	00000	00000	00000	00000
int → *func*	01111	00000	00000	00000	00000
func → *int*	00000	10000	10000	10000	10000
func → *func*	00000	01111	01111	01111	01111

Fig. 10.65. O valor de $\tau m \sigma$.

[illegible]

Um Sistema Simples de Tipos

Para ilustrar as ideias por trás dos nossos algoritmos de inferência de tipos, introduzimos um sistema simples de tipos e uma linguagem baseada no Exemplo [illegible]. Os enunciados de nossa linguagem são de três tipos.

1. [illegible]
2. [illegible]
3. [illegible]

Inferimos os tipos realizando uma análise de fluxo de dados no grafo de fluxo de um programa consistindo de enunciados desses três tipos. [illegible]

[illegible]

O Esquema para Adiante

[illegible]

1. [illegible]
2. [illegible]
3. [illegible]

Formalmente,

[illegible]

[illegible]

O Esquema para Trás

[illegible]

1. [illegible]
2. [illegible]
3. [illegible]

[illegible]

antes do conjunto [illegible] a menos que o comando [illegible] seja [illegible]. Por essa razão, podemos computar os conjuntos pelas regras seguintes, relativamente a [illegible]. Notemos, em segunda, que, como nas regras para adiante, podemos fazer interseção a partir do laço de que os tipos de [illegible] precisam ser compatíveis antes do comando. No caso [illegible], podemos também restringir os tipos em [illegible] uma atribuição analoga e válida a respeito de [illegible]. Por outro lado, se [illegible], os tipos de [illegible] após o comando são irrelevantes; os tipos de [illegible] antes do comando são formados de maneira que nenhuma restrição desse tipo é permitida. De novo, uma atribuição analoga a [illegible]. Finalmente, determinemos um mapeamento especial, usando nomes [illegible], de forma a refletir essa decisão. Por exemplo, definimos

[illegible]

Agora podemos restringir [illegible] aqueles tipos que são incompatíveis com os conjuntos de tipos [illegible]. Ao mesmo tempo, podemos restringir os tipos de [illegible] descartando as [illegible] que não possuem tipos compatíveis. Isso [illegible] produzirá um tipo que [illegible] possa ter. Por conseguinte, definimos

[illegible]

Antes de prosseguir para o algoritmo de determinação de tipos, vejamos rapidamente de novo a discussão das definições inoperantes na Seção 10.5 que, se começarmos com a falsa suposição de que alguma definição *d* esteja disponível em algum ponto num laço, poderemos propagar erroneamente essa hipótese ao longo do laço, mesmo que certa computação com um conjunto de definições precedentes mais do que o necessário. Um problema similar ocorre na determinação de tipos, onde a suposição de que uma variável possa ter um certo tipo "previsto" às vezes se mostra um problema em que avaliações em todo o laço. Por conseguinte, vamos introduzir um progresso terceiro valor, adicionalmente aos $\mathcal{N}$ conjuntos de tipos e parte do Exemplo 10.51, que um mapeamento pode atribuir a uma variável o valor *indefinido*. Esse valor de *task finish* é similar a seu uso na estrutura de propagação de constantes da seção prévia.

Durante a avaliação, o valor *indefinido* possui qualquer outro valor, como de costume [illegible]. Para um dado ponto, nos referimos aos tipos de [illegible] como tipos, por exemplo, no comando [illegible]. A variável [illegible] também mudar qualquer um conjunto de tipos, isso é [illegible]. Por conseguinte, os tipos, por exemplo, o valor de uma variável [illegible] é [illegible]. Esse [illegible] de [illegible] é [illegible] a seu uso na estrutura de propagação de constantes da seção prévia.

Algoritmo 10.19

Entrada. Um grafo de fluxo cujos blocos são [illegible] e cujos tipos numéricos [illegible].

Saída. Um conjunto de tipos para cada variável em cada ponto. O conjunto de tipos [illegible] contém no máximo um tipo [illegible] e [illegible] tipos [illegible] por ponto.

Método. Computamos os mapeamentos [illegible] e [illegible] para cada bloco B. Cada mapeamento representa as suposições do programa a respeito de tipos nos vários pontos [illegible] do Exemplo 10.51. Inicialmente, todos os mapeamentos retornam cada variável para *indefinido*.

Realizamos, então, passagens alternadas para adiante e para trás através do grafo de fluxo, até que passagens consecutivas para adiante e para trás ambas falhem em alterar qualquer mapeamento. A passagem para adiante é realizada por

para cada bloco B examinado em ordem de profundidade
faça início
  [illegible]
  [illegible]
fim

A passagem para trás é

para cada bloco B examinado em ordem reversa de profundidade
faça início
  [illegible]
  [illegible]
fim

Exemplo 10.52. Consideremos o programa simples sem ramificações mostrado na Fig. 10.66. Estamos interessados em saber o tipo apropriado que designamos μ_1 a μ_4. Cada [illegible] B_i, quando [illegible] B_i. Tecnicamente, não é assumido que B tenha dois antecessores, pois assumimos que os blocos separadamente [illegible] não [illegible]. No entanto, não estamos preocupados com isso que ocorre antes do [illegible] B, porque todas as variáveis podem ter qualquer tipo lá.

Segue-se que precisamos de quatro passagens antes que a [illegible] não [illegible] ocorra e outras duas para detectar que [illegible] não ocorrerá. As passagens são sumarizadas na Fig. 10.67(a)–(e). Na primeira passagem e para adiante. Quando [illegible] B_1, descobrimos que [illegible] pode ser do tipo inteiro, porque é usada [illegible] mapeamento [illegible]. Descobrimos, também, que [illegible] é [illegible] e, subsequentemente, [illegible] pode supostamente ser mapeada para [illegible] em μ_1 e μ_2. Essas [illegible] são sumarizadas na Fig. 10.67(a).

A segunda passagem, mostrada na Fig. 10.67(b), é para trás. Nesta passagem, quando consideramos B_4, sabemos que [illegible] precisa ser [illegible] [illegible] foi aplicado a [illegible]. Por isso, [illegible] pode [illegible] ser [illegible]. Na terceira passagem, a qual é para adiante, esta restrição de tipo de [illegible] propaga-se por [illegible] [illegible] [illegible], como mostrado em 10.67(c).

A quarta passagem é para trás, como mostrado na Fig. 10.67(d). Aqui, o fato de que [illegible] é um argumento de [illegible] em B_3 nos diz que [illegible] pode ser somente um inteiro. Igualmente, quando avaliarmos, em B_2, encontramos que o resultado de [illegible] pode somente ser do tipo de [illegible]. Esses fatos possibilitam que [illegible] seja do tipo [illegible]. Finalmente, na Fig. 10.67(e), vemos como a quinta passagem para adiante esses fatos [illegible] se propagam. Nenhuma [illegible]

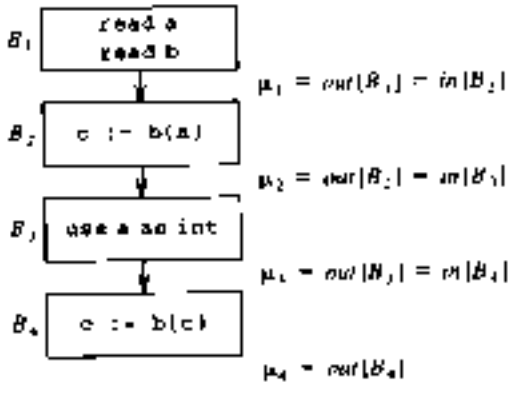


Fig. 10.66. Programa exemplo.

	a	b	c
μ_1	11101	11013	*undef*
μ_2	11111	01010	10101
μ_3	00001	01010	11101
μ_4	00000	01111	11101

(a) Para A_1 e A_2

	a	b	c
μ_1	10000	01100	*undef*
μ_2	10000	01111	11111
μ_3	10000	01101	11111
μ_4	10000	01111	11111

(b) Para T_1 e T_2

	a	b	c
μ_1	10000	01001	*undef*
μ_2	10000	01100	11111
μ_3	10000	01100	11111
μ_4	10000	01100	11111

(c) Para A_3 e A_4

	a	b	c
μ_1	10001	01000	*undef*
μ_2	10000	01100	10000
μ_3	10000	01100	10000
μ_4	10000	01100	11111

(d) Para T_3 e T_4

	a	b	c
μ_1	10000	01000	*undef*
μ_2	10000	01000	10000
μ_3	10000	01000	10000
μ_4	10000	01000	10000

(e) Para A_5 e A_6

Fig. 10.67. Simulação do Algoritmo de Euclides para o código da Fig. 10.66.

nomes, não podem ser feitas novas atribuições. Neste caso, reduzimos os conjuntos de possíveis tipos a um único para cada variável a cada ponto. É também interessante diminuir o número de nomes diferentes. Digamos que podemos usar dois nomes para cada variável em cada ponto.

10.13 DEPURAÇÃO SIMBÓLICA DE CÓDIGO OTIMIZADO

Um *depurador simbólico* é um sistema que nos permite examinar os dados do programa enquanto o mesmo está sendo executado. O depurador é usualmente chamado quando ocorre um erro de programa, ou como um recurso de capacitação, ou quando certos comandos do usuário são encontrados pelo programador no código-fonte. Uma vez invocado, o depurador simbólico permite que o programador examine e possivelmente modifique quaisquer das variáveis que são correntemente acessíveis pelo programa em execução.

Para um comando do usuário como "exibir o valor corrente de x", o depurador precisa ter disponível certas informações:

1. Deve haver uma forma de se associar um identificador como x à localização que o mesmo representa. Por conseguinte, a parte da tabela de símbolos que é associada com x deve conter a localização, por exemplo, um local numa área de dados global ou um registro de ativação para algum procedimento. Essa informação é gerada pelo compilador e passada para o depurador. Essas informações, por exemplo, podem ser codificadas dentro do código para o programa.

2. Deve haver informações de escopo, de forma a podermos tornar inambíguas as referências a um identificador que é declarado diversas vezes em diferentes procedimentos. Lembre-se que os dados locais a um procedimento existem apenas enquanto o procedimento está sendo executado, e o depurador precisa ser capaz de encontrar os dados apropriados em tempo de execução. De novo, essas informações precisam ser obtidas a partir da tabela de símbolos do compilador e passadas para o depurador, por exemplo, junto com o código.

3. Precisamos saber onde estamos no programa quando o depurador for invocado. Essas informações são inseridas pelo compilador no código para o depurador, quando o comando para a invocação para o depurador, declarado pelo usuário. Também obtida a partir do tratador de exceções quando um erro em tempo de execução faz com que o depurador seja chamado.

4. A fim de que as informações de localização do programa sejam úteis, é preciso ser capaz de mapear entre os identificadores do código-fonte e as localizações de dados correspondentes. Isso também é feito pelo compilador, com base na tabela de símbolos.

Um depurador precisa de um grande cuidado para ser escrito. Ele precisa considerar as dificuldades que podem ocorrer se o usuário esquecer um identificador ou bobear ao escrever um comando. À primeira vista, pode parecer que não há necessidade de se depurar um programa otimizado. Sob o ponto de vista de desenvolvimento, o modo de operação depurar-e-otimizar não é compatível, não otimizante, será usado até que o sistema se torne razoavelmente livre de erros. Somente então um compilador otimizante será usado.

Infelizmente, um programa pode realizar corretamente com um compilador não otimizante e falhar com os mesmos dados de entrada quando processado pelo compilador otimizante. Por exemplo, pode haver um erro de compilador otimizante, através do reordenamento das operações. Um compilador otimizante pode introduzir um espaço de capacidade ou gerar um resultado imprevisível em precisão da máquina. Inclusive, os compiladores otimizantes podem realizar algumas transformações simples, como a eliminação de subexpressões comuns locais ou reordenar o código dentro de um bloco básico, o que faz uma grande diferença na dificuldade de se projetar um depurador simbólico. Para os

seguinte. Precisaremos considerar que algoritmos e estruturas de dados usados em depuradores simbólicos para um compilador otimizante que transforma blocos básicos de formas arbitrárias.

Deduzindo Valores de Variáveis em Blocos Básicos

Por uma questão de simplicidade, vamos assumir que tanto a versão do código-fonte quanto a do objeto sejam seqüências de enumerados intermediários. O tratamento do código-fonte como código intermediário, porém, apresenta problemas uma vez que o código é mais geral do que o preciso. Por exemplo, o usuário pode ser autorizado a solicitar valores em locais chamadas para o depurador entre os enumerados-fonte, mas aqui permanecerá que as chamadas sejam colocadas após qualquer otimização de código intermediário. O tratamento do código objeto como código intermediário é questionável enquanto se otimizador quebrar tais códigos em blocos básicos em diversos enumerados de maneira que quedem separados. Por exemplo, por alguma razão, o compilador pode compilar os dois enumerados intermediários

```
[illegible]
```

em outros, digamos, de duas adições seguidas realizadas em registradores diferentes e encadeadas. Se esse for o caso, podemos criar as cargas e armazenamentos de registradores como se os registradores fossem locais de armazenamento manipulados pelo código intermediário, como, por exemplo

```
[illegible]
```

Diversos problemas ocorrem ao interagirmos com o usuário a respeito de um bloco, onde o usuário pensa que o código-fonte está sendo executado, mas de fato uma versão otimizada daquele bloco está sendo:

1. Vamos supor que estejamos executando um programa que resulte da otimização de alguns blocos básicos do programa-fonte, e digamos que o usuário peça a execução do enumerado [illegible] e queira investigar o estado de uma variável [illegible]. Precisamos interpretar a nome [illegible] que ocorre em um dos enumerados-fonte. Uma vez que [illegible] pode ser uma sobrepressão comum figurando em dois ou mais enumerados-fonte, para que contexto [illegible] o erro?

2. Um problema mais difícil ocorre se o usuário do depurador deseja ver o valor "correto" de alguma variável [illegible]. Num programa otimizado, [illegible] pode ter recebido uma atribuição por último em algum enumerado [illegible]. Mas no programa-fonte, [illegible] pode ser apenas enumerado ao qual o depurador foi invocado, de forma que o valor de [illegible] que está disponível para o depurador nesse ponto não é o valor que o usuário pensa que seja o valor "correto" de [illegible] de acordo com a linguagem do código-fonte. Similarmente, [illegible] pode prover valores enumerados que não existem no código-fonte. Existe uma única atribuição a [illegible] entre os mesmos, de forma que o valor de [illegible] disponível para o depurador estará desatualizado. É possível formar disponível o valor correto para o usuário? Por exemplo, poderia ser o valor de alguma outra variável na versão otimizada daquele bloco ser computado a partir dos valores de outras variáveis?

3. Finalmente, se o usuário estabelecer uma chamada para o depurador após algum enumerado do código-fonte, quando o conjunto de declarações deve ser dado ao depurador durante a execução do código otimizado?

```
(1)  c := a+b      (1')  d := a+b
(2)  d := c        (2')  t := b+e
(3)  e := c-e      (3')  a := d-e
(4)  a := d-e      (4')  b := d/t
(5)  b := b+e      (5')  c := a
(6)  b := d/b
       (a)                (b)
```

Fig. 10.68. Código-fonte e otimizado.

Uma solução seria notar a versão otimizada de cada bloco básico junto com a versão otimizada, de forma a tornar disponível o valor correto de cada variável em todos os instantes. Registramos esta "decisão" porque os erros mais sutis e especialmente aqueles introduzidos pelo compilador podem desaparecer quando as instruções que constituem o problema forem separadas umas das outras no tempo ou no espaço.

A solução que adotamos é providenciar informações sobre os valores respeitantes cada bloco básico, de forma que o depurador possa responder à questão: "É possível prover [illegible] o valor correto de uma variável [illegible] em um enumerado [illegible]?" A estratégia que usamos para prover estas informações é o GDA de blocos básicos, anotado com informações a respeito de que variáveis guardam o valor correspondente a variáveis do GDA, e em que tempos, tanto no programa-fonte quanto no otimizado. A notação

```
[illegible]
```

anotada a um nó significa que o valor representado por aquele nó armazenado numa variável [illegible] no ponto de término do enumerado [illegible] do código-fonte, exatamente antes de ser [illegible] executado. Se [illegible] for omitido, o valor será válido até o fim do bloco.

Exemplo 10.53. Na Fig. 10.68(a), vemos um bloco básico de um código-fonte e na Fig. 10.68(b) está uma possível versão otimizada daquele código. A Fig. 10.69 mostra o GDA para o bloco, com indicações dos valores nos quais as variáveis guardam esses valores, tanto no código-fonte quanto no otimizado. Os apóstrofos são usados para indicar que os enumerados estão no código otimizado. Os exemplos não anotados [illegible] são os do código-fonte, a partir do início do enumerado 2 até exatamente depois da atribuição para [illegible] em 3. Também o valor de [illegible] no código-fonte, a partir do início do enumerado 3 até o final. Adicionalmente, o interesse reside no valor de [illegible] no código otimizado a partir do enumerado 2' até o final. [illegible]

Agora podemos responder à primeira questão levantada acima. Suponhamos que o usuário peça o valor de uma variável [illegible] enquanto se executa o enumerado 2' do código otimizado. Uma vez que o mesmo valor será computado por qualquer enumerado após o qual

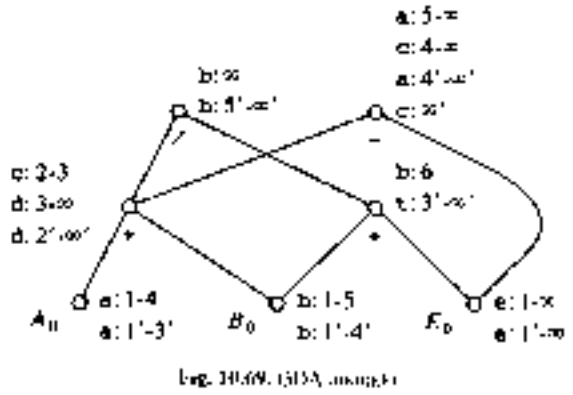


Fig. 10.69. GDA anotado.

o mesmo nó do GDA que o enunciado j'. Faz sentido reportar ao usuário que um erro ocorreu no primeiro enunciado-fonte a computar este nó. Por conseguinte, no Exemplo 10.53, se um erro ocorresse no enunciado 1', 2', 3' ou 4', reportaríamos que o mesmo teria ocorrido nos enunciados 1, 3, 3 e [illegible] respectivamente. Nenhum erro pode ocorrer no enunciado 5', porque nenhum valor é computado. Postergamos os detalhes sobre como os enunciados correspondentes são computados até o Exemplo 10.54, a seguir.

Podemos também responder à segunda questão. Suponhamos que estejamos no enunciado j' do código otimizado e o usuário diz que o controle está no enunciado i do código-fonte. [illegible] Se o usuário solicitar o valor da variável x, precisamos encontrar uma variável [illegible] (frequentemente, mas nem sempre, [illegible]) tal que o valor de x no enunciado i do código-fonte esteja no mesmo nó do GDA que [illegible] no enunciado j' do código otimizado. Inspecionamos o GDA para o nó que representa o valor de x em i e podemos ler daquele nó todas as variáveis do programa objeto que alguma vez tiveram aquele valor, para ver se uma delas abriga este valor em j'.

Se assim o for, estamos feitos, senão, podemos ainda computar o valor de x em i a partir de outras variáveis em j'. Seja n o nó para x no tempo i. Podemos, então, considerar os filhos de n, digamos m e p, para ver se algum desses nós representa o valor de alguma variável no tempo j'. Se negarmos existir uma variável para m, mas nenhuma para p, podemos considerar os filhos de p recursivamente. Em última análise, ou encontramos uma forma de computar o valor de x no instante i, ou concluímos que não existe uma tal forma de computá-lo. Se encontramos uma forma de computar os valores de m e de p, então os computamos e em seguida aplicamos o operador de n para computar x no instante i.[1]

Exemplo 10.54. Suponhamos que durante a execução do código da Fig. 10.68(b) um erro ocorra no enunciado 2'. O enunciado está computando o nó rotulado $*$ na Fig. 10.69 e o primeiro enunciado-fonte computando aquele valor é o enunciado 5. Reportamos, consequentemente, um erro no enunciado 5.

Na Fig. 10.70, tabulamos os nós do GDA ao início dos enunciados 5 e 2', respectivamente, os nós são indicados por seus rótulos, quer um símbolo de operador ou um símbolo de valor inicial, como d_0. Indicamos também, como computar o valor no tempo 5 a partir dos valores das variáveis no tempo 2'. Por exemplo, se o usuário solicitar o valor de [illegible] o valor do nó rotulado [illegible] é fornecido. Nenhuma variável possui aquele valor no tempo 2', mas, felizmente, existem variáveis [illegible] que guardam o valor de cada um dos filhos do nó [illegible] no instante 2', de forma que podemos computar o valor de [illegible] computando o valor de [illegible].

Agora, vamos responder à terceira questão: como tratar as chamadas do depurador inseridas pelo usuário. Num certo sentido, a resposta é trivial. Se o usuário faz uma chamada ao depurador após o enunciado i do programa-fonte, podemos parar a execução do programa ao início do bloco. Se o usuário deseja ver o valor de alguma variável x após o enunciado i, consultamos o GDA anotado para ver que nó representa o valor desejado e o computamos aquele valor a partir dos valores iniciais das variáveis para aquele bloco.

Por outro lado, pode nos deixar menos trabalho para o depurador e também evitar algumas situações onde as tentativas para computar um valor levam a erros que necessitam ser anunciados ao usuário, se postergarmos as chamadas para o depurador para um tempo tão tarde quanto possível. É fácil computar o último enunciado j' do programa otimizado, de forma que chamamos o depurador após j' e enganamos para o usuário que a chamada foi feita após o enunciado i do programa-fonte. Para encontrar j', seja S o conjunto de nós do GDA que corresponde ao valor de alguma variável do programa-fonte imediatamente após o enunciado i. Podemos ser solicitados pelo usuário para computar qualquer valor em S. Por conseguinte, podemos chamar o depurador após o enunciado j' do código otimizado somente se, para cada nó n em S, existir algum $k' \geq j'$, tal que alguma variável esteja associada ao nó n no tempo k' no código otimizado. Para tanto, sabemos que o valor de n ou está disponível imediatamente após o enunciado j' ou será computado algum tempo após o enunciado j'. No primeiro caso, é trivial computar o valor de n se chamarmos o depurador após j', enquanto que, no último caso, sabemos que os valores disponíveis após j' são suficientes para computar n de alguma forma.

Variável	Valor		Obtido de
	Tempo 2'	Tempo 5	
[illegible]	4	[illegible]	[illegible]
[illegible]	B	t_6	[illegible]
[illegible]	indefinido	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	E,	t_5	[illegible]
[illegible]	indefinido		

Fig. 10.70. Valores das variáveis nos tempos 2' e 5.

Exemplo 10.55. Consideremos de novo os códigos-fonte e otimizado da Fig. 10.68, e suponhamos que o usuário insira uma chamada ao depurador após o enunciado 3 do código-fonte. Para encontrar o conjunto S inspecionamos o GDA da Fig. 10.69 e vemos que nós possuem variáveis do programa-fonte atreladas aos mesmos no tempo 4. Esses nós são aqueles rotulados [illegible] naquela figura.

Em seguida, examinamos de novo o GDA com a finalidade de encontrar o maior j' tal que cada um dos nós em S tenha alguma variável do código otimizado atrelada a si num tempo estritamente superior a j'. Os nós rotulados [illegible] não apresentam problemas, uma vez que seus valores são carregados pelas variáveis [illegible], respectivamente, num tempo [illegible]. Os nós [illegible] e [illegible] efetivamente fornecem o valor de [illegible] e o que perde seu valor mais cedo, dentre os dois, é [illegible], cujo valor é destruído pelo enunciado 3'. Por conseguinte, $j' = 2$ é o maior valor possível de j', isto é, se o usuário solicita uma chamada ao depurador após o enunciado-fonte 3, podemos concedê-la após o enunciado 2'.

O leitor deve estar atento para uma sutileza no Exemplo 10.55, para a qual não há realmente uma boa solução. Se rodarmos o código otimizado através do enunciado 2' antes de chamarmos o depurador, um erro no cômputo de [illegible] no enunciado 2', por exemplo, ao [illegible] pode fazer com que o depurador seja chamado antes da chamada pretendida. No entanto, como o cômputo correspondente aos enunciados 2' não ocorre até o enunciado 5 no programa-fonte, iremos informar ao usuário que um erro ocorreu no enunciado 5. Será um tanto misterioso para o usuário a forma com que chegamos ao enunciado 5 sem chamar o depurador no enunciado 3. Provavelmente, a melhor solução para este problema é não permitir que j' seja tão grande que exista um enunciado k' no código otimizado, com $k' \leq j'$, tal que o código-fonte não compute o valor computado por k' até depois do enunciado i, no qual a chamada para o depurador foi colocada.

Efeitos da Otimização Global

Quando nosso compilador realiza otimizações globais, existem problemas mais difíceis para o depurador simbólico resolver e, frequentemente, não existe nenhuma forma de encontrar o valor correto de uma va-

[1] [illegible]

[illegible]

Eliminação de Variáveis de Indução

[illegible]

[illegible]

Eliminação Global de Subexpressões Comuns

[illegible]

[illegible]

Movimentação de Código

[illegible]

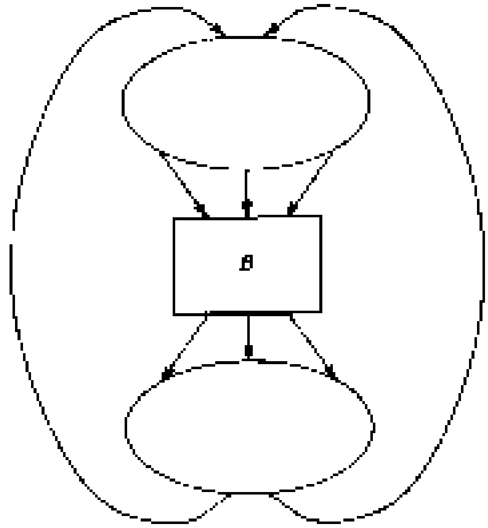


Fig. 10.71. [illegible]

[illegible]

[illegible]

[illegible]

1. [illegible]

2. [illegible]

3. [illegible]

[illegible]

[illegible]

sendo usados, seria razoável, então, inserir código no programa otimizado, que usa a classe independente de máquina, a versão otimizada que seguimos através do Capítulo 10, uma solução que melhoramos mais tarde nas sugestões de leitura adicional, porque interferia com o código otimizado.

EXERCÍCIOS

10.1 Considere a multiplicação matricial na Fig. 10.72.

a) Suponha que a, b e c tenham, cada um, n elementos inteiramente e que haja quatro bytes por palavra numerada consecutivamente a partir de 0. Produza os endereços de base para o programa da Fig. 10.72.

b) Gere o código de máquina-alvo a partir dos enumerados de três endereços.

c) Divida em grafo de blocos básicos a parte dos enumerados de três endereços.

d) Encontre os subexpressões comuns a partir de cada bloco básico.

e) Encontre os laços no grafo de fluxo.

f) Mova as computações invariantes para fora dos laços.

g) Encontre as variáveis de indução para cada laço e as elimine onde for possível.

h) Gere o código da máquina-alvo a partir do grafo de fluxo em (g). Compare o código com aquele produzido em (b).

10.2 Compute as definições incidentes e cadeias-ud para o grafo de fluxo original do Exercício 6. Encontre o grafo de fluxo final a partir de (d) logo.

10.3 O programa da Fig. 10.73 conta os primos de 1 até n, usando o método da "peneira". Seja n um valor suficientemente grande.

a) Organize o programa da Fig. 10.73 em quíntuplos de três endereços, estimando o número total de bytes em sua memória alocada estaticamente.

b) Gere o código da máquina-alvo a partir dos enumerados de três endereços.

c) Construa um grafo de fluxo para os enumerados de três endereços.

d) Mostre a árvore de dominadores para o grafo de fluxo em (c).

e) Para o grafo de fluxo em (c), indique os laços reflexivos e seus laços naturais.

f) Mova as computações invariantes para fora dos laços usando o Algoritmo 10.7.

g) Elimine as variáveis de indução onde quer que seja possível.

h) Propague para frente os enumerados de cópia sempre que possível.

i) É possível uma quebra de laço? Se sim, faça-a.

j) Na suposição de que n seja sempre par, simplifique os laços mais internos um de cada vez. Que novas otimizações são agora possíveis?

```
begin
    read n;
    for i := 2 to n do
       a[i] := true;   /* initialize */
    count := 0;
    for i := 2 to n ** .5 do
       if a[i] then   /* i is a prime */
          begin
             count := count + 1;
             for j := 2 * i to n by i do
                a[j] := false
            /* j is divisible by i */
          end;
    print count
end
```

Fig. 10.73. Programa para calcular primos.

10.4 Repita o Exercício 10.3 na suposição de que n seja tão grande que a memória alocada dinamicamente, com n bytes, tenha de ser usada para a própria palavra-chave.

10.5 Para o grafo de fluxo da Fig. 10.74, compute:

a) as cadeias-ud e du

b) as variáveis vivas no final de cada bloco

c) as expressões disponíveis.

10.6 É possível algumas das posições para constante na Fig. 10.47? Se for fácil-o.

10.7 Há subexpressões comuns na Fig. 10.74? Se houver, elimine-as.

10.8 Uma expressão e deve ser movida para um ponto p em um programa que pretendemos seja tomado a partir de e as expressões são avaliadas antes que qualquer uma delas seja operada ou definida. Forneça um algoritmo, do tipo descrito na Seção 10.6, para encontrar todas as expressões candidatas a tais. Indique que operador de conjunção $\wedge$ deveria usar e se a propagação para frente ou para trás é preferível. Use o algoritmo em seu grafo de fluxo da Fig. 10.74.

***10.9** Seja expressão e movível para o ponto p se e for disponível em p e for computada em p e preservando seu valor para usos subsequentes. (Note esta otimização, que é claramente não recomendável em geral, mas pode produzir ganhos de espaço.) Forneça um algoritmo para calcular expressões movíveis computadas.

10.10 Existem expressões que podem ser alocadas na Fig. 10.74? Se houver, elimine-as.

10.11 Onde possível, propague para frente quaisquer passos de cópia introduzidos nas modificações dos Exercícios 10.6, 10.7 e 10.10.

10.12 Um *bloco básico estendido* é uma sequência de blocos básicos $B_1, \ldots, B_n$, tal que, para $1 < i \leq n$, B_i é o único predecessor de B_i e B_i não é um único predecessor. Encontre o final do bloco básico estendido a cada nó:

a) da Fig. 10.39

b) do grafo de fluxo construído no Exercício 10.1(e)

c) da Fig. 10.74.

***10.13** Forneça um algoritmo que, dado um grafo de fluxo, encontre o final do bloco básico estendido a cada nó.

10.14 Podemos realizar alguma otimização em blocos básicos estendidos que não é possível em blocos básicos? Forneça algoritmos para realizar as seguintes otimizações de nível de bloco básico estendido em cada caso, indique que efeitos podem ser, sobre outros blocos básicos estendidos, e dê um exemplo dentro de um bloco básico estendido.

```
begin
    for i := 1 to n do
       for j := 1 to n do
          c[i,j] := 0;
    for i := 1 to n do
       for j := 1 to n do
          for k := 1 to n do
             c[i,j] := c[i,j] + a[i,k] * b[k,j]
end
```

Fig. 10.72. Programa de multiplicação matricial.

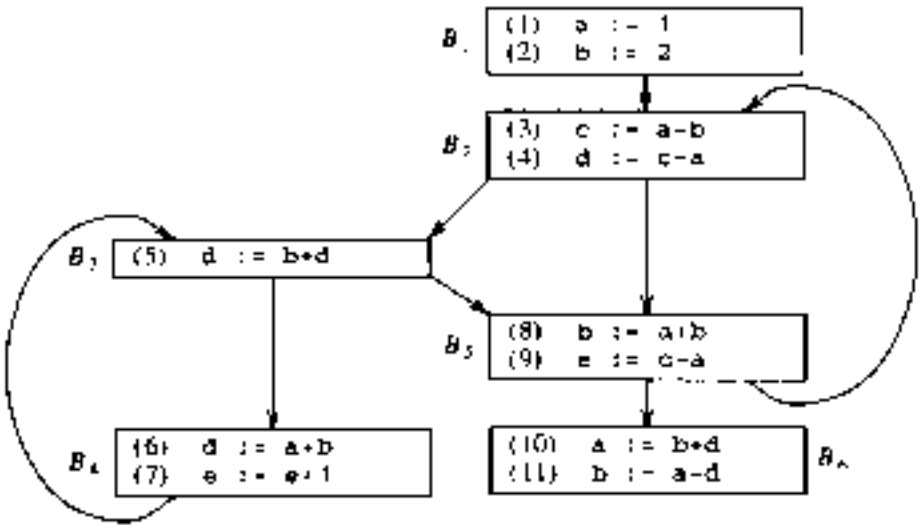


Fig. 10.74. Grafo de fluxo.

a) Eliminação de subexpressões comuns.
b) Transposição para constantes.
c) Propagação de cópias.

10.15 Para o grafo de fluxo do Exercício 10.13 faça:
a) Encontre uma sequência de reduções T_1 e T_2 para o grafo.
b) Encontre a sequência de graus de intervalo.
c) Qual é o grafo de fluxo limite? É o grafo de fluxo redutível?

10.16 Repita o Exercício 10.15 para o grafo de fluxo da Fig. 10.34.

**10.17* Mostre que as seguintes condições são equivalentes (são as definições alternativas de "grafo de fluxo redutível"):
a) O limite de reduções T_1, T_2 é um único nó.
b) O limite da análise de intervalos é um único nó.
c) O grafo de fluxo pode ter seus lados divididos em duas classes — ditos lados avançados e retrocessos, os lados de "retorno", constituindo lados cujas cabeças dominam suas caudas.
d) O grafo de fluxo não possui subgrafo da forma mostrada na Fig. 10.75. Aqui, a é o nó inicial e b, c e d são nós arbitrários distintos com a exceção que $a = d$ é possível. As setas representam caminhos não-vazios de nós distintos (exceto para os pontos finais, naturalmente).

10.18 Forneça algoritmos para computar as expressões disponíveis e as variáveis vivas para a linguagem com apontadores descrita na Seção 10.8. Assegure-se de fazer suposições conservativas a respeito de variáveis, *aliases* e desvios entre nós.

10.19 Use um algoritmo para computar as definições interprocedurais atingindo incidentes, usando o modelo da Seção 9.8. De novo assegure-se de fazer as suposições conservativas da verdade.

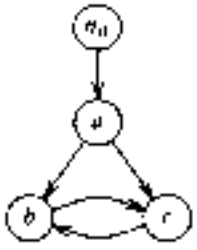


Fig. 10.75. Subgrafo proibido para grafos de fluxo redutíveis.

10.20 Suponha que a transmissão de parâmetros seja por valor e não por referência. Podem esses dois nomes ser usados um pelo outro? E se for usada a ligação de cópia e restauração (transmissão por valor-resultado)?

10.21 Qual é a profundidade do grafo de fluxo do Exercício 10.13?

**10.22* Prove que a profundidade de um grafo de fluxo redutível nunca é maior do que o número de vezes que análise de intervalos precisa ser realizada de forma a produzir um único nó.

**10.23* Generalize o algoritmo de análise de fluxo de dados baseado em estruturas da Seção 10.8 para uma estrutura geral de fluxo de dados no sentido da Seção 10.11. Que suposições a respeito de ϕ e $\cdot$ você terá que fazer para assegurar que seu algoritmo funciona?

**10.24* Uma estrutura interessante para o cálculo de expressões disponíveis é obtida supondo-se os "valores" a serem propagados sejam todas as possíveis partições de expressões de tal forma que duas expressões estão na mesma classe se e somente se têm o mesmo valor. Para evitar ter que instanciar a infinidade de possíveis expressões, podemos representar os valores listando somente as expressões não-triviais que sejam equivalentes a alguma expressão. Por exemplo, se executarmos os seguintes

```
        :
    x := y + z
        :
```

então teremos as seguintes equivalências imediatas: $x = y + z$, $x - y = z$, $x - z = y$. A partir dessas segue-se que todas as equivalências $x + y = x + z$, $x + y = y + z$, $x + y = x + x$, $x + y = y + y$, $x + y = z + z$, $x + y = 2x$, $x + y = 2y$, $x + y = 2z$ são também válidas. Assim, não há necessidade de listar explicitamente.
a) Qual é a representação apropriada de reunião ou confluência para esta estrutura?
b) Forneça uma estrutura de dados para representar os valores e os algoritmos para implementar os operadores de transição.
c) Quais são as funções apropriadas para associar com os nós do fluxo? Explique o efeito que a função associada às atribuições, tais como $x := y + z$, deve ter na variável x.
d) A estrutura é distributiva? Monotônica?

10.25 Como você usaria os dados capturados pela estrutura do Exercício 10.24 para realizar:
a) eliminação de subexpressões comuns?
b) propagação de cópias?
c) transposições para constantes?

10.26 Suponha que a transmissão de parâmetros seja por valor e não por referência. Podem esses dois nomes ser usados um pelo outro? E se for usada a ligação de cópia e restauração (transmissão por valor-resultado)?

10.27 Qual é a profundidade do grafo de fluxo do Exercício 10.13?

**10.28* Prove que a profundidade de um grafo de fluxo redutível nunca é maior do que o número de vezes que análise de intervalos precisa ser realizada de forma a produzir um único nó.

**10.29* Generalize o algoritmo de análise de fluxo de dados baseado em estruturas da Seção 10.8 para uma estrutura geral de fluxo de dados no sentido da Seção 10.11. Que suposições a respeito de ϕ e $\cdot$ você terá que fazer para assegurar que seu algoritmo funciona?

**10.30* Uma estrutura interessante para o cálculo de expressões disponíveis é obtida supondo-se os "valores" a serem propagados sejam todas as possíveis partições de expressões de tal forma que duas expressões estão na mesma classe se e somente se têm o mesmo valor. Para evitar ter que instanciar a infinidade de possíveis expressões, podemos representar os valores listando somente as expressões não-triviais que sejam equivalentes a alguma expressão. Por exemplo, se executarmos os seguintes

```
        :
    x := y + z
        :
```

então teremos as seguintes equivalências imediatas: $x = y + z$, $x - y = z$, $x - z = y$. A partir dessas segue-se que todas as equivalências $x + y = x + z$, $x + y = y + z$, $x + y = x + x$, $x + y = y + y$, $x + y = z + z$, $x + y = 2x$, $x + y = 2y$, $x + y = 2z$ são também válidas. Assim, não há necessidade de listar explicitamente.
a) Qual é a representação apropriada de reunião ou confluência para esta estrutura?
b) Forneça uma estrutura de dados para representar os valores e os algoritmos para implementar os operadores de transição.
c) Quais são as funções apropriadas para associar com os nós do fluxo? Explique o efeito que a função associada às atribuições, tais como $x := y + z$, deve ter na variável x.
d) A estrutura é distributiva? Monotônica?

10.31 Como você usaria os dados capturados pela estrutura do Exercício 10.30 para realizar:
a) eliminação de subexpressões comuns?
b) propagação de cópias?
c) transposições para constantes?

*10.26 Forneça provas formais de seguinte a respeito da relação < sobre prioridades:
a) $x < x$ é falso (irreflexividade).
b) $x < y$ implica $y < x$ é falso (assimetria).
c) $x < y$ e $y < z$ implica $x < z$ (transitividade).
d) $x < y$ e $y < z$ implica $x = y$.

**10.27 Mostre que as seguintes condições são necessárias e suficientes para o algoritmo de travessia de árvore de dados, executado recursivamente em profundidade, converter a árvore em "n" mais a multiplicidade" passagens. Para todas as funções e e g e valores v:

$$h(e(v), h(f(v), h(g(v), v))) = h(f(v), h(e(v), h(g(v), v)))$$

10.28 Mostre que as estruturas de definições independentes de expressões disponíveis satisfazem a condição do Exercício 10.27. *Nota:* De fato, essas estruturas convergem em 3 mais a profundidade passagens.

**10.29 Essa condição do Exercício 10.27 implica a pela mencionada condição? Pela distributividade? E vice-versa?

10.30 Vemos na Fig. 10.76 dois blocos básicos. O primeiro avaliza m e n? E o segundo, uma versão otimizada.
a) Construa GID's para os blocos da Fig. 10.76(a) e (b). Verifique que, na suposição de semente = estado inicial, esses dois blocos são equivalentes.
b) Aplique GIDA com os tempos nos quais o valor de cada variável é conhecido a cada nó.
c) Verifique, para um erro que ocorra a cada um dos pontos indicados (1, 2, ..., 6) que ocorrem no fluxograma da Fig. 10.76(a), quais devem ser os testes que a mesma ocorra.
d) Para cada um dos erros do item (c), indique para que variáveis da Fig. 10.76(a) é possível computar o valor e como fazemos isso.
e) Vamos supor que nos permitimos usar variáveis globais (normalmente válidas como "Seja $x = 0$; se então $x = 1$ ..."). A sua resposta acima ainda é mantida?

10.31 Generalize o Exemplo 10.14 para que leve em conta um conjunto arbitrário de entidades *known*. Em linhas gerais, para cada nó, encontramos o lugar mais cedo em que todos os caminhos que chegam a ele possam ser diretamente para a próxima iteração do laço. Sugestão: Use as técnicas descritas nos livros da Seção 10.10 para os grafos de fluxo reduzíveis.

10.32 Mostre que no Algoritmo 10.3 os conjuntos de definições em *avail* e *sold* podem ser escritos. Semelhantemente, mostre que no Algoritmo 10.4 esses algoritmos podem ser escritos.

10.33 Generalize o Algoritmo 10.9 para eliminação de variáveis de indução para o caso onde as constantes multiplicativas podem ser negativas.

10.34 Generalize o Algoritmo 10.9 para determinar para que os apontadores possam apontar a parte da Seção 10.8, para casos onde os apontadores são usados somente para apontar para outros apontadores.

*10.35 Assuma que cada um dos seguintes conjuntos infinitos seja estimativas muito grandes ou muito pequenas são conservadoras? Explique sua resposta em termos do uso pretendido das informações.
a) Expressões disponíveis.
b) Variáveis modificadas por um procedimento.
c) Variáveis não modificadas por um procedimento.
d) Variáveis de indução não modificadas por um procedimento.
e) Variáveis de indução pertencentes a uma dada família.
f) Dependências de ordem que atingem um dado ponto.

*10.36 Reformule o Algoritmo 10.12 para que compute os predecessores de uma dada variável em um dado ponto.

*10.37 Modifique o Algoritmo 10.12 para os casos em que são dados os seus parâmetros:
a) por valor.
b) por cópia e restauração (valor-resultado).

*10.38 Prove que o Algoritmo 10.15 converge para um ponto-fixo não necessariamente próximo das variáveis totalmente modificadas.

*10.39 Generalize o Algoritmo 10.13 para que determine as variáveis modificadas em casos em que as variáveis do tipo procedimento sejam permitidas.

*10.40 Prove que, em cada grafo de intervalos, cada nó representa uma região do grafo de fluxo original.

10.41 Prove que o Algoritmo 10.16 computa corretamente o conjunto de dominadores de cada nó.

*10.42 Modifique o Algoritmo 10.17 (definições incidentes) para usar dados em estruturas para computar as definições incidentes para pequenas regiões designadas apenas sem requerer que todo o grafo de fluxo esteja presente na memória de uma vez. Certifique-se de que seu resultado é conservativo. Adapte sua técnica para as expressões disponíveis. Que outras propriedades e termos e informações úteis?

*10.43 Na Seção 10.6 propusemos um algoritmo de aceleração usando uma combinação de uma redução T_1 com uma redução T_2. Prove a correção desta modificação.

10.44 Generalize o método gráfico da Seção 10.11 para problemas de fluxo para trás.

**10.45 Prove que, quando o Algoritmo 10.18 converge, a solução resultante é a única solução, mostrando que, para cada percurso P de comprimento n após n iterações, $meet\{f^n(t) \mid t \in P\}$.

10.46 Na Fig. 10.77 está um grafo de fluxo de um programa na linguagem hipotética usada na Seção 10.12. Encontre a melhor estimativa dos tipos de cada variável usando o Algoritmo 10.19.

NOTAS BIBLIOGRÁFICAS

Informações adicionais a respeito de otimização de código podem ser encontradas em Cocke e Schwartz [1970], Aho e Ullman [1973], Schaefer [1973], Hecht [1977], Jones e Muchnick [1981], Allen [1975] provide uma introdução para a otimização de programas.

Muitos compiladores otimizantes são descritos na literatura. Lowry e Medlock [1969] discutem um compilador precursor de técnicas de otimização sofisticadas. Lowry e Medlock [1969] e Scarborough e Kolsky [1980] descrevem a otimização de compiladores FORTRAN. Busam e Englund [1969] e Morel et al. [1982] descrevem heurísticas adequadas para otimização em Fortran. Wulf et al. [1975] discute o projeto de um influente compilador otimizante para Bliss.

Allen et al. [1986] descreve um sistema que foi construído para experimentar otimizações de programas. Cocke e Markstein [1980] reporta a respeito da eficiência de várias otimizações para uma linguagem semelhante a PL/I. Archibald, Cytron, Steinmetz e MacLaren [1982] descrevem a implementação de transformações conhecidas que foram usadas em compiladores para PL/I e C. Auslander e Hopkins [1982] reportam a respeito de um compilador para uma variante de PL/I que usa um algoritmo simples para produzir um código intermediário em baixo nível e o otimizado por transformações gráficas. Freudenberger,

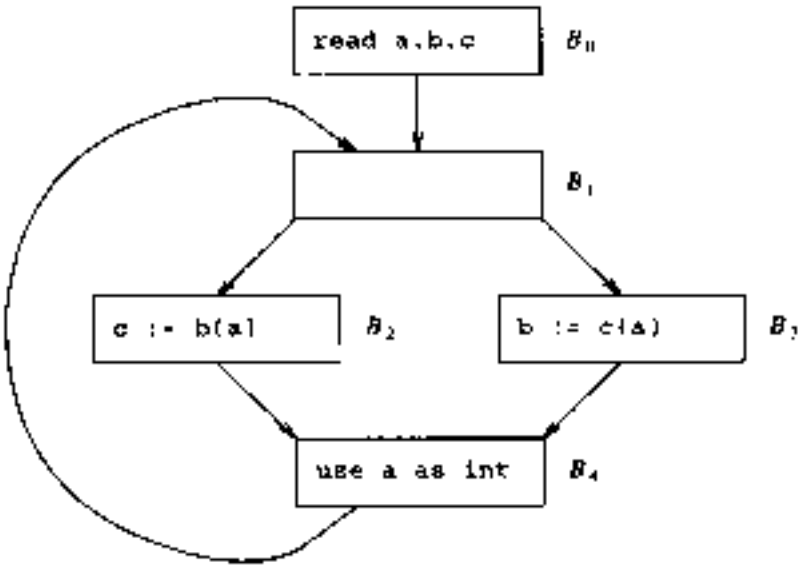


Fig. 10.27. Programa exemplo para detecção de tipos

Schwartz e Shaw [1983] descrevem experiências com um reconhecedor para SETL. Chow [1984] relata experimentos com um reconhecedor portável para Pascal, independente da máquina. Powell [1984] descreve um compilador reconhecedor portável independente da máquina para Modula-2.

O estudo sistemático das técnicas de análise de dados começou com Kildall [1973] e Cocke [1970] desde então publicados juntos como Allen e Cocke [1976], apesar de vários métodos de análise de fluxo de dados existirem em versões anteriores dessa época.

A análise de fluxo de dados dirigida pela sintaxe, como apresentada na Seção 10.5, foi baseada em Bliss (Wulf et al. [1975]). Giegerich [1982], SIMPL-Z (Bowen e Reid [1973]) e Madsen 2 (Powell [1984]). Discussões adicionais desta família de algoritmos encontram-se em Hecht e Schaefer [1978], Hecht [1977] e Rosen [1977].

A abordagem iterativa à análise de fluxo de dados, descrita na Seção 10.6, foi pesquisada por Vyssotsky (ver Vyssotsky e Wegner [1963]) e que inseriu o método num compilador Fortran. O uso de sofisticadas técnicas em potencialidade para melhorar a eficiência é proveniente de Hecht e Ullman [1975].

A abordagem da análise de intervalos para a análise de fluxo de dados foi pioneirizada por Cocke [1970]. Kennedy [1971] originalmente usou a análise de intervalos para problemas de fluxo para trás, como as variáveis vivas. Existe razão para se acreditar, segundo Kennedy [1976], que os métodos baseados em intervalos são um tanto mais eficientes do que os iterativos, se a linguagem que está sendo compilada tende a produzir poucos grafos de fluxo não redutíveis, se algum. A variante usada aqui baseada em T_1 e T_2 é proveniente de Tarjan [1973]. Uma versão ainda mais rápida que tira vantagem do fato de que a maioria das regiões tem uma única saída foi descrita por Graham e Wegman [1976].

A definição original de um grafo de fluxo redutível, aquele que se transforma em um único nó pela análise de intervalo iterada, é proveniente de Allen [1970]. Caracterizações equivalentes são encontradas em Hecht e Ullman [1972, 1974], Kasyanov [1973] e Tarjan [1974a]. A conversão de nós para grafos não redutíveis é de Cocke e Miller [1969].

A idéia de que o fluxo de controle estruturado é mantido por grafos de fluxo redutíveis é expressa em Kosaraju [1974]. Bansal, Prasanna e Rokne [1973] e Cherniavsky, Henderson e Kershaw [1976]. Baker [1977] descreve seus uso em um algoritmo de estruturação de programas.

A abordagem livre de ciclos aplicada à análise de fluxo de dados começou com Kildall [1973], Tenenbaum [1974] e Weg-

man [1975] são fundações similares. A versão discutida no Algoritmo de Kildall na qual a ordem em profundidade é usada é proveniente de Kam e Ullman [1976].

Enquanto Kildall assumia a condição de distributividade para suas estruturas, tais como a de composição de constantes (ex. Exemplo 10.42), não atendendo ao fato da adequação da monotonicidade foi percebida num número de *papers* que lidam com algoritmos de fluxo de dados, tais como Tenenbaum [1974], Schwartz [1975a,b], Graham e Wegman [1976], Kam e Ullman [1976] Kam e Ullman [1977] e Cousot e Cousot [1977].

Uma vez que algoritmos diferentes requerem diferentes suposições a respeito dos dados, a teoria sobre que propriedades são necessárias para certos algoritmos foi desenvolvida por Kam e Ullman [1977], Rosen [1980] e Tarjan [1981].

Uma direção que se seguiu a partir do *paper* de Kildall foi a melhoria de algoritmos para se lidar com problemas particulares de fluxo de dados (por exemplo o Exemplo 10.42) que ele mesmo introduziu. Uma dessas idéias é a de que o reticulado de elementos não precisa ser totalmente ordenado, mas que pode ser explorada a fato de que são realmente mapeamentos de variáveis para valores. Ver Reif e Lewis [1977] e Wegman e Zadeck [1985]. Kou [1977] também explora a idéia para problemas mais convencionais.

Kennedy [1981] é um levantamento das técnicas de análise de fluxo de dados e Cousot [1981] faz uma pesquisa extensiva dos diferentes territórios a respeito dos reticulados.

Graham [1975] introduziu as otimizações baseadas de fluxo de não-orientação de código e uma forma limitada de eliminação de variáveis de indução. Allen [1969] e outro trabalho fundamental em otimização de laços; Allen e Cocke [1972] e Wulfe [1976a] são pesquisas mais extensivas das técnicas na área. Morel e Renvoise [1979] descrevem um algoritmo que elimina simultaneamente cômputos redundantes e invariantes dos laços.

A discussão da eliminação das variáveis de indução na Seção 10.7 está baseada em Loveman e Medlock [1969]. Ver Allen, Cocke e Kennedy [1981] para algoritmos mais poderosos.

Um algoritmo para alguns dos problemas de laços mais discutidos aqui, tais como encontrar se existe um percurso de a até b que passe pelo menos uma vez por c, pode ser resolvido por um algoritmo eficiente de Weyman [1983].

Quase todos os compiladores, tanto para a conveniência de laços quanto para realizar a otimização de código, foi primeiramente com Loveman e Medlock [1969]. Lesk e Barnes [1976] descrevem o Portal [1979]

O Algoritmo 10.1 para se encontrar os dominadores foi descoberto independentemente por Purdom e Moore [1972] e Aho e Ullman [1973a]. O conhecimento de uma ordenação em profundidade para acelerar o algoritmo é proveniente de Hecht e Ullman [1975], embora tenha sido a forma assintoticamente mais eficiente de realizar a tarefa é proveniente de Tarjan [1974a]. Lengauer e Tarjan [1979] descrevem um algoritmo mais eficiente para encontrar os dominadores, o qual é adequado para uso prático.

O estudo da predominância e da análise intervalar começou com Spillman [1971] e Allen [1974]. Foram desenvolvidos alguns métodos mais poderosos do que aqueles da Seção 10.8. Um geral é o clássico artigo de Cocke e Kennedy sobre algoritmos de programação de forma a evitar alguns dos piores de pseudônimos imprevisíveis que ocorrem algumas vezes em simples "descuidos". Esses trabalhos incluem Barth [1978], Banning [1979] e Weihl [1980]. Veja também Ryder [1979] na construção de grafos de chamada.

Um forma sumária da análise interprocedimental é obtida através de expressões regulares de fluxo de dados de programas é discutido por Henry [1984].

O papel fundamental sobre os tipos de análise de fluxo de dados é Kennedy [1974], no qual nossa discussão sobre a Seção 10.12 está baseada. Kaplan e Ullman [1980] fornecem um algoritmo mais poderoso para a detecção de tipos.

A idéia associada de dependência condicional de código otimizador na Seção 10.13 é proveniente de Hennessy [1982].

Tem havido um número de papéis que tentam avaliar a melhoria devido a otimizações. O valor de uma otimização parece ser altamente dependente da linguagem que está sendo compilada. O leitor pode desejar consultar o estudo clássico de otimização de Fortran em Knuth [1971b] ou os papéis de Cajewska [1975], Palm [1975], Cocke e Kennedy [1976], Cocke e Markstein [1980], Chow [1983], Chow e Hennessy [1984] e Powell [1984].

Outro tópico na otimização que não cobrimos aqui é a otimização de linguagens "em nível muito alto". Isso está na linguagem (ou pérola de conjuntos SETL, onde mudamos realmente os algoritmos subjacentes e as estruturas de dados. Uma otimização central nesta área é a eliminação generalizada de variáveis de indução, como em Earley [1974b], Fong e Ullman [1976], Paige e Schwartz [1977], e Jonge [1984].

Outro passo-chave na otimização de linguagens de nível muito alto é a seleção das estruturas de dados: este tópico é coberto em Schwartz [1975a, b], Low e Rovner [1976] e Schonberg, Schwartz e Sharir [1981].

Não recompilamos, também, rotinas durante a otimização incremental de código, onde pequenas modificações no programa não requerem uma completa reotimização. Ryder [1983], Cooper e Kennedy [1984] e Burke e Cytron [1986] tentam realizar a otimização incremental de blocos básicos.

Para terminar, devemos mencionar algumas das muitas outras formas sobre as quais as técnicas de análise de fluxo de dados têm sido usadas. Hecht e Ullman [1984] usam-nas para provar transições entre dois analisadores sintáticos para realizar a recuperação de erros. Harrison [1977] e Suzuki e Ishihata [1977] discutem sobre a verificação dos limites de arrays em tempo de compilação.

Um dos usos mais significativos da análise de fluxo de dados foi a da otimização de código estendida, através da verificação estática de tempo de compilação de erros. Fosdick e Osterweil [1976] é um papel fundamental, enquanto que Osterweil [1981], Adrion, Brenstad e Cherniavsky [1982] e Brenstad [1984] introduzem alguns dos desenvolvimentos mais recentes.

CAPÍTULO 11

DESEJA ESCREVER UM COMPILADOR?

Neste capítulo, examinamos os principais tópicos técnicos e de gerenciamento para o projeto de compiladores. Supondo que você deseje escrever um compilador, ou tenha algum projeto em andamento, ele provavelmente será implementado de forma mais rápida e eficiente se você seguir algumas das idéias apresentadas neste capítulo. Também veremos algumas questões de implementação que surgem na construção de compiladores. Muito da discussão aqui está baseada na experiência de compiladores usando o sistema operacional UNIX e suas ferramentas.

11.1 PLANEJANDO UM COMPILADOR

Um bom compilador pode estar dividido em processadores para a linguagem-fonte e para o gerador de código-alvo. Usando a estrutura global resultante obtemos, em alguns casos, um projeto para um compilador que consiste em um conjunto de módulos. Vários fatores determinam o projeto e a implementação desses módulos.

Temas das Linguagens-Fonte

O "tamanho" de um compilador afeta o tamanho e número dos módulos. Apesar de não existir definição precisa para o tamanho de uma linguagem, é claro que um compilador para uma linguagem como Ada ou PL/I é maior e mais difícil de implementar do que um compilador para uma pequena linguagem como Rational compile-processador Fortran "racional". Kernighan [1975] cita BDN como uma linguagem para a composição tipográfica de textos matemáticos.

Um fator importante para a extensão em que a linguagem-fonte evolui ou muda no longo curso da construção do compilador. Apesar da especificação da linguagem-fonte parecer imutável, poucas linguagens permanecem inalteradas durante a vida útil de um compilador. Mesmo as linguagens amadurecidas evoluem, apesar de lentamente. Fortran, por exemplo, mudou consideravelmente desde o padrão da linguagem que era em 1957; os laços, as expressões DO e os comandos condicionais em Fortran 77 são bem diferentes daqueles na linguagem original. Rosler [1981] relata a evolução de C.

Por outro lado, uma linguagem nova, experimental, pode ser alterada sempre que uma nova idéia que seja implementada. Uma forma de se criar uma nova linguagem é escrever um compilador de um pré-processador para uma linguagem já existente. Muitas necessidades de certos grupos de usuários. Muitas dessas "pequenas" linguagens, descritas originalmente no sistema UNIX, tais como AWK e EQN, foram criadas desta forma.

Conseqüentemente, um escritor de compilador poderia antecipar pelo menos um certo número de modificações na definição da linguagem fonte, ao longo do tempo de vida do compilador. O projeto modular e o uso de ferramentas podem auxiliar o escritor do compilador a lidar com essas mudanças. Por exemplo, usando-se geradores para analisadores léxicos e sintáticos em um compilador, pode-se aplicar muitas das mesmas ferramentas às modificações e mudanças na definição da linguagem mais prontamente do que quando ambos os componentes são inteiramente no código.

Os Temas das Linguagens-Alvo

A natureza das limitações e da linguagem-alvo também pode ter efeito sobre um compilador. Especificamente, é preciso levar em consideração uma série de fatores ao projetar um compilador e suas estratégias de geração de código que deverão ser usadas. Se a linguagem-alvo for nova, o escritor do compilador é aconselhado a assegurar que a mesma está correta e que suas sequências de temporização são bem compreendidas. Caso contrário, ao tentar escrever um compilador, pode ser difícil saber se o compilador é culpado ou se a máquina-alvo pode ser a culpada. Também é importante ter um bom depurador de código-objeto para a máquina-alvo. Além disso, compiladores recentes estão incluindo a capacidade de depurar algum papel. Concomitantemente, o projeto de linguagens intermediárias é importante, na medida em que auxilia detalhes específicos de máquina para um pequeno número de módulos.

Critérios de Desempenho

Existem vários aspectos de desempenho do compilador: velocidade, qualidade do código, diagnósticos de erro e condições de manutenção. As linguagens e suas especificações podem ter efeito sobre todos esses parâmetros independentemente. Por exemplo, a velocidade de compilação é mais importante do que a velocidade do código-objeto? Quão importantes são os diagnósticos e a recuperação de erros?

A velocidade do compilador pode ser melhorada através da redução do número de passagens. Isso é muitas vezes possível, talvez até o ponto de se gerar uma linguagem de máquina diretamente

em uma única passagem. Nesse caso, esse tipo pode produzir um compilador que não serve um código-alvo de uma qualidade razoável para uso pessoal de Lisp. (Veja o Capítulo 12.)

Existem dois aspectos da portabilidade: reorientabilidade e reutilizabilidade. Um compilador reorientável é aquele que pode ser modificado facilmente de forma a gerar código para uma nova linguagem-alvo. Um compilador reutilizável é aquele que pode ser facilmente movido de forma a que rode numa nova máquina. Um compilador portável pode ser usado sem modificação para um projetado para uma máquina específica, porque um compilador para uma única máquina pode seguramente se basear em suposições específicas sobre a máquina-alvo que um compilador portável não pode.

11.2 ABORDAGENS PARA O DESENVOLVIMENTO DOS COMPILADORES

Existem vários enfoques diferentes que o autor de um compilador pode adotar para implementá-lo. A forma mais simples é recriar ou reutilizar um compilador existente. Se não existir um compilador adequado, o autor pode adotar a seguinte abordagem: escrever um compilador para uma linguagem similar e implementar os componentes correspondentes, usando ferramentas de geração de compiladores ou implementá-los à mão. É relativamente raro que uma organização de compilador completamente nova seja requerida.

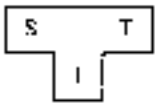
Não importa que enfoque seja adotado, escrever compiladores é um exercício em engenharia de *software*. As idéias a partir de muitos estudos na área de *software* (veja, por exemplo, Brooks [1975]) podem ser aplicadas para melhorar a confiabilidade e as condições de manutenção do produto final. Um projeto que acolha mudanças prontamente era permitir que o compilador evolua para uma linguagem. O uso de ferramentas de construção de compiladores pode ser um auxílio significativo nesse aspecto.

Bootstrapping

Um compilador é um programa complexo o suficiente para que desejemos escrevê-lo numa linguagem mais amigável do que uma linguagem de montagem. No ambiente de programação UNIX, os compiladores são usualmente escritos em C. Mesmo os compiladores C são escritos em C. Quais as facilidades oferecidas por uma linguagem para escrever a si mesma é conseqüência do *bootstrapping*. Aqui nos examinamos o uso do *bootstrapping* para criar compiladores em pilhas de uma máquina para outra através da modificação da interface de retaguarda. As idéias básicas sobre o *bootstrapping* são muito antigas, desde meados dos anos 50 (Strong et al. [1958]).

O *bootstrapping* pode levantar a questão, "como foi compilado o primeiro compilador?", que sua própria conta, naquele momento, a galinha ou o ovo? Uma é mais fácil de responder. Para uma resposta, considere Lisp. Se você tem uma linguagem de programação, McCarthy [1981] nota que, no final de 1958, Lisp foi usada como uma notação para escrever funções e elas foram traduzidas à mão em linguagem de montagem e executadas. A implementação de um interpretador para Lisp escrita nessa linguagem por McCarthy descreve mostra que Lisp era uma notação para descrever funções "mais ajustada do que as máquinas de Turing ou as definições recursivas usuais na teoria das funções recursivas"[1] e, por conseguinte, escrever uma função definida *eval[e,a]* em Lisp, que tomava uma expressão Lisp e como argumentos. S. R. Russell observou que *eval* poderia servir como um interpretador para Lisp codificado à mão e, conseqüentemente, com uma linguagem de programação com um interpretador. Como mencionado na Seção 11.1, através de operar um interpretador, que interpretado efetivamente realiza as operações de programa fonte.

Para os propósitos de *bootstrapping*, um compilador é caracterizado por três linguagens: a linguagem fonte S que o mesmo compila, a linguagem-alvo T para a qual gera código e a linguagem I de implementação em que é escrito. Representamos as três linguagens pelo seguinte diagrama, chamado de *diagrama-T*, por causa de sua forma (Bratman [1961]).



De modo abreviado, abreviamos o diagrama-T acima como S|T|I. As três linguagens S, I e T podem ser todas completamente diferentes. Por exemplo, um compilador pode rodar numa máquina e produzir código-alvo para uma outra máquina. Tal compilador é tipicamente chamado de *compilador cruzado*.

Vamos supor que escrevemos um compilador utilizável para uma nova linguagem L. Vamos implementá-lo em linguagem de montagem N, para gerar código para a máquina N existente. Temos L|N. Se um compilador existente para a máquina M é para gerar código para M é conhecido como S|M|M. Se L|N é executado através de S|M|M, obtemos o compilador L|M|N, isto é, um compilador de L para N que roda em M. Este processo é ilustrado na Fig. 11.1, colocando-se juntos os diagramas-T para esses compiladores.

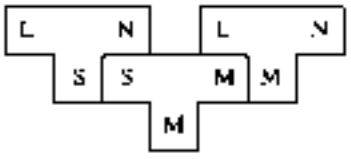


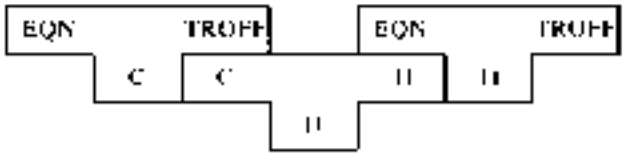
Fig. 11.1. Compilação cruzada.

Quando os diagramas-T são colocados juntos, como na Fig. 11.1, a linguagem de implementação S do compilador L|N precisa ser a mesma que a linguagem fonte do compilador existente S|M|M e a linguagem-alvo M do compilador existente precisa ser a mesma que a linguagem de implementação da forma tradicional L|N. Uma forma de diagramas-T, tais como a da Fig. 11.1, pode ser pensada como a equação

$$L|N = S|M|M + L|M|N$$

Exemplo 11.1. A primeira versão do compilador EQN (veja a Seção 8.2.1) foi escrita em C e usou a linguagem de implementação e pré-processador para o formatador de textos TROFF. Como mostrado no diagrama seguinte, um compilador cruzado para EQN, rodando num PDP-11, foi obtido rodando-se EQN/C TROFF através do compilador C|11|11 do PDP-11.

Uma outra forma de *bootstrapping* constrói um compilador para sub-conjuntos cada vez maiores de uma linguagem. Suponhamos que uma

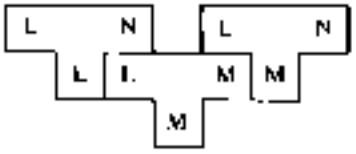


[1] Veja os exemplos. Todas as funções recursivas primitivas são computáveis, mas não é preciso saber nada sobre recursão primitiva para escrever um interpretador para Lisp. Veja também Kleene [1952] para uma discussão sobre funções recursivas primitivas e computabilidade. Veja também Rosser [1939] para uma discussão sobre a relação entre Lisp e o cálculo lambda de Church [1941].

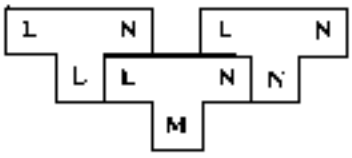
nova linguagem L a ser implementada na máquina M. Como primeiro passo, pode-se escrever um pequeno compilador que traduza um subconjunto S de L (o mínimo possível) para M; isto é, um compilador $S_{S}[M$. Em seguida, usamos o subconjunto S para escrever um compilador $L_{S}M$ para L. Quando $L_{S}M$ é traduzido através de $S_{S}[M$, obtemos uma implementação de L, $L_{S}M$. Pelo menos uma das primeiras linguagens a ser implementada em sua própria linguagem (Huskey, Halstead e McArthur [1960]).

Wirth [1971] nota que Pascal foi primeiro implementada escrevendo-se um compilador numa pequena linguagem Pascal. O compilador foi então traduzido "à mão" numa linguagem de baixo nível disponível sem qualquer tentativa de otimização. O compilador era para um subconjunto "limpo" de Pascal: vários estágios de *bootstrapping* mais tarde, um compilador todo de Pascal foi obtido. Lecarme e Peyrolle-Thomas [1978] sumarizam métodos que foram usados para realizar o *bootstrapping* de compiladores Pascal.

Para que as vantagens do *bootstrapping* sejam realizadas integralmente, um compilador processado em uma linguagem em que ele mesmo é escrito é necessário. Vamos supor que escrevemos um compilador $L_{S}N$, para a linguagem L em L, de forma a gerar código para a máquina N. O desenvolvimento pode ter lugar na máquina M, onde um compilador existente $L_{S}M$ para L roda e gera código para M. Compilando-se primeiro $L_{S}L$ com $L_{S}M$ obtemos um compilador cruzado $L_{S}N$ que roda em M, mas produz código para N:



O compilador $L_{S}N$ pode ser compilado uma segunda vez, desta vez usando o compilador cruzado gerado:



O resultado desse segundo estágio é um compilador $L_{N}N$, que roda em N e gera código para N. Existe um número de aplicações úteis deste processo em duas etapas, de forma que o veremos novamente (mais adiante, Fig. 11.2).

Exemplo 11.2. Este exemplo é motivado pelo desenvolvimento do compilador Fortran H (veja a Seção 12.4). O compilador foi, ele próprio, escrito em PL/I e passou por um processo de *bootstrapping* três vezes. A primeira vez foi para converter do IBM 7094 para o Sistema 360 — uma tarefa árdua. A segunda vez para otimizar a si mesmo, o que reduziu o tamanho do compilador de aproximadamente 550K para algo em torno de 410K bytes (Conway e Medlock [1969]).

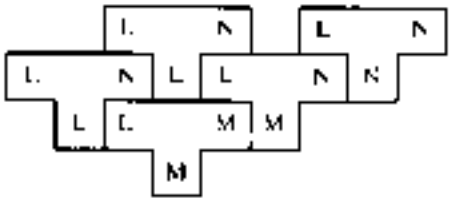
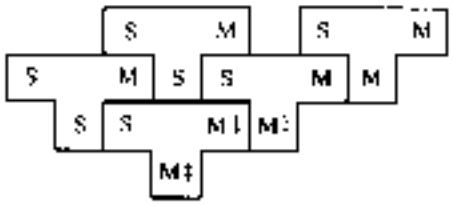


Fig. 11.2 Reduzindo o tamanho de um compilador

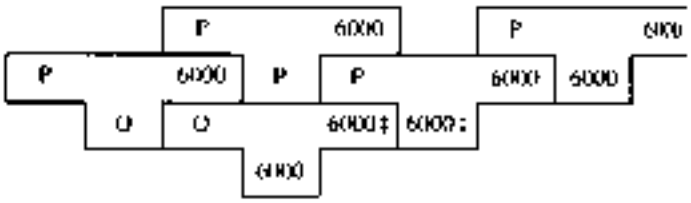
Usando as técnicas de *bootstrapping*, um compilador só muito raramente poderá otimizar a si próprio. Suponhamos que todo o desenvolvimento seja feito na máquina M. Temos $S_{S}M$, um bom compilador otimizante para a linguagem S, escrito em S, e desejamos $S_{S}M$, um bom compilador otimizante para S, escrito em M.

Podemos criar $S_{M}M$, um rápido e "sujo" compilador para S em M que não gera um código de baixa qualidade, mas também leva muito tempo para fazê-lo. M rodando uma pobre implementação em M $S_{M}M$ é uma implementação pobre de um compilador que gera código pobre. No entanto, podemos usar o compilador indiferente $S_{M}M$ para obter um bom compilador para S e todas as etapas



Primeiro, o compilador otimizante $S_{S}M$ é traduzido pelo compilador rápido e "sujo" para produzir $S_{M}M$, uma pobre implementação do compilador otimizante mas que produz código de boa qualidade. O bom compilador otimizante $S_{S}M$ é então recompilado-se $S_{S}M$ através de $S_{M}M$.

Exemplo 11.4. Atkinson [1981] descreve como uma implementação "limpa" de Pascal foi obtida num processo similar àquele do Exemplo 11.2. As versões de Pascal foram à venda de um compilador novo em 1972, para as máquinas da série CDC 6400. Novos graus, segundo, O (previsto) Pascal (antigo) e Pascal linguagem revisada



Um compilador para o Pascal revisado foi escrito em Pascal e otimizado pelo próprio P(antigo)6400. Como no Exemplo 11.2, o símbolo ":" marca uma fonte de ineficiência. O compilador velho não gerou, ou não gerou eficiente o bastante. Por conseguinte, a velocidade do compilador $[P_{(6400)}\_6000]$ era um tanto moderada e seus engenheiros de memória bastante altas (Atkinson [1981, p. 4]). As versões em Pascal eram pequenas o suficiente para que o compilador P(6400) pudesse ser traduzido e rodado com pouca relação ao P(6400) e rodasse através do compilador mutante P_{6400} 6000 para obter uma implementação "limpa".

11.3 O AMBIENTE DE DESENVOLVIMENTO DE COMPILADORES

Num sentido real, um compilador é somente um programa. O ambiente no qual esse programa é desenvolvido pode afetar quão rápida e confiavelmente o mesmo é implementado. A linguagem na qual o compilador é implementado é igualmente importante. Apesar de os compiladores terem sido escritos em linguagens como Fortran, uma escolha mais recente para a implementação de compiladores é uma linguagem orientada para sistemas, tal como C.

Se a linguagem-fonte, por si só, for uma nova linguagem orientada para sistemas, faz muito sentido escrever um compilador na sua

...ntem. linguagem. Usando as técnicas de bootstrapping, discutida na seção anterior, compila-se o compilador com ele mesmo e depois recompila-se novamente.

As ferramentas de construção de compiladores fornecem um ambiente de programação poderoso, facilitando grandemente a criação de um compilador eficiente. Ao se escrever um compilador é útil se particionar o programa todo em módulos, onde cada módulo pode ser processado de formas bastante distintas. Um programa que gerencie o processamento desses módulos é uma ajuda indispensável ao escritor do compilador. O sistema UNIX oferece um comando chamado *make* [Feldman 1979a] que gerencia o número muito grande de arquivos que vão formar um programa de computador, onde controla-se relacionamentos entre os módulos de programa e onde apenas aqueles que foram modificados precisam ser recompilados para formar os módulos executáveis após as modificações terem sido realizadas.

Exemplo 11.4. O comando *make* lê a especificação das tarefas que precisam ser realizadas a partir de um arquivo chamado [illegible]. No Seção 2.4 construímos um compilador através da compilação de sete arquivos com um compilador C. Cada um deles depende de um arquivo de definições chamado *globals.h*. Para montar o compilador a tarefa de juntar as partes do compilador pode ser feita por um *makefile* como o seguinte, supondo que o compilador resultante se chame *cc*. A especificação de [illegible] poderia se parecer com:

```
[illegible]
```

Quando digitamos a primeira linha, faz com que [illegible] à esquerda ligue os sete arquivos objeto à direita. As linhas longas podem ser partidas colocando-se um *backslash*, imediatamente antes da parte interrompida. As duas partes na segunda linha dizem que [illegible] a esquerda depende de todos os arquivos à direita. Tal linha de dependências pode ser seguida por um comando para construir o alvo à esquerda, a partir dos arquivos à direita. A terceira linha, portanto, diz que o programa *cc* é criado ligando-se os arquivos objeto [illegible]. Lembramos que o *make* só irá recompilar os arquivos objeto fonte correspondentes [illegible] e compilando cada um com o compilador C, para criar os arquivos objeto correspondentes. A última linha [illegible] diz que todos os sete arquivos objeto dependem do arquivo global de cabeçalho [illegible].

O *makefile* é usado diretamente se simplesmente o comando *make* for dado. Também é possível se especificar comandos alternativos, [illegible]

```
[illegible]
```

Subsequentemente, uma compilação será refeita somente se um arquivo dependente for mais recente que o arquivo após a última compilação. Kernighan e Pike [1984] contém exemplos do uso de *make* para facilitar a construção de um compilador.

Um gerador de parser é comumente o mais importante para a escrita de compiladores. Uma vez que um compilador inteiro está escrito, um gerador de perfis pode ser usado para determinar onde o compilador está gastando seu tempo, o lugar em que compila um programa comum. A identificação e modificação dos pontos críticos pode muitas vezes dar ao compilador um fator de dez ou mais.

Adicionalmente às ferramentas de software, um número de ferramentas tem sido criado especificamente para o processo de desenvolvimento de compiladores. Na Seção 3.5, descrevemos o gerador Lex, que pode ser usado para produzir automaticamente um analisador léxico a partir de uma especificação sobre a forma de expressões regulares, na Seção 4.9, descrevemos o gerador Yacc, que pode ser usado para produzir automaticamente um analisador sintático LR a partir de uma descrição gramatical da sintaxe da linguagem. Obviamente descrevemos ambas ferramentas porque elas são muito úteis. Adicionalmente aos geradores de analisadores léxicos e sintáticos, os geradores de gramáticas de atributos e de código têm sido criados para auxiliar a construção dos componentes de compiladores. Muitas dessas ferramentas de construção de compiladores possuem a desejável propriedade de que elas são baseadas em especificações dos mesmos.

Tem havido algum debate a respeito da eficiência de compiladores que são de geradores de programas na construção de compiladores. Waite e Carter [1985] observaram que geradores de programas bem implementados se constituem numa ajuda significativa na produção de componentes confiáveis de compiladores. Frequentemente tal produz um analisador sintático correto usando uma descrição gramatical da linguagem e um gerador de analisadores sintáticos do que implementar manualmente o analisador diretamente a mão. Um fator importante, entretanto, é quão bem esses geradores interfaceiam-se naturalmente com os outros programas. Um erro comum no projeto de um gerador é assumir que o mesmo será o único no projeto. Um projeto melhor destina o gerador a produzir software com rotinas inteligentes que possam ser chamadas por outros programas [Johnson e Lesk 1978b].

11.4 TESTE E MANUTENÇÃO

Um compilador tem que gerar código correto. Idealmente, gostaríamos que o compilador certificasse matematicamente que o compilador implementou tecnicamente suas especificações. Vários artigos teoricamente discutem a criação dos vários algoritmos de compilação mas infelizmente, os compiladores raramente são especificados de tal forma que uma implementação abstrata possa ser matematicamente verificada em conjunto com uma especificação formal. Como os compiladores são atualmente feitos um outro objetivo existe também o teste de se checar se a própria especificação também está correta.

Na prática precisamos nos voltar para algum método sistemático de teste de um compilador, de forma que atinjamos a nossa confiança em que o mesmo irá funcionar satisfatoriamente no campo. Um enfoque usado com sucesso por muitos escritores de compiladores é o teste de "regressão". Aqui mantemos um conjunto de programas de teste e sempre que o compilador for modificado, os programas de teste são compilados usando tanto a versão nova do compilador quanto a antiga. Qualquer diferenças nos programas objeto produzidos pelos dois compiladores são sinalizadas ao escritor do compilador. O comando *cmp* do sistema UNIX pode ser usado para automatizar o teste.

A escolha dos programas para incluir no conjunto é um problema difícil. Idealmente, gostaríamos que os programas de teste executassem cada comando dentro do compilador pelo menos uma vez. Isso provê, e requer, uma grande cobertura de teste. Conjuntos de teste existem também para várias linguagens (ver, por exemplo, [ACM 1972]). Muitos escritores de compiladores adicionam ao teste de regressão programas que explorem falhas nas versões anteriores de seus compiladores, e frustrante ter um velho problema ressurgindo por causa de uma nova correção.

O teste de desempenho também é importante. Alguns escritores de compiladores verificam se as novas versões do compilador geram

simplesmente que seja apresentada um bom número de versões previamente testadas através da realização de testes de medição de tempo como parte do teste de regressão.

A manutenção de um compilador é um caso especial. Um compilador particularmente seu compilador vai ser usado em ambientes diferentes com o pessoal envolvido no projeto do compilador original. Um elemento crítico para tornar mais fácil a manutenção de um compilador é um bom estilo de programação e uma boa documentação. Os autores recomendam um compilador que foi escrito usando-se somente sete comandos, muitos dos quais se lia "Este código está errado". Desnecessário dizer que um tal programa é difícil de ser mantido por qualquer um que não seja o autor original.

Knuth [1984b] desenvolveu um sistema chamado WEB que combina o problema de documentação de grandes programas escritos em Pascal. WEB facilita a programação literária, a documentação e desenvolvimento incremental de um código, tudo em um processo posterior de levantamento. Muitos dos utilitários em WEB podem ser aplicados igualmente bem a outras linguagens.

CAPÍTULO 12

UM RÁPIDO EXAME DE ALGUNS COMPILADORES

Este capítulo discute a estrutura de alguns compiladores existentes para uma linguagem de formatação de texto, Pascal, C, Fortran, Bliss e Modula-2. Nossa intenção não é advogar projetos apresentados aqui ou determinado de outros, mas, ao invés, ilustrar a variedade que é possível na implementação de um compilador.

Os compiladores para Pascal foram escolhidos porque influenciaram o projeto da própria linguagem. Os compiladores para C foram escolhidos porque C é a linguagem-padrão de programação no sistema operacional UNIX. O compilador Fortran H foi escolhido porque influenciou significativamente o desenvolvimento das técnicas de otimização. BLISS/11 foi escolhida para ilustrar o projeto de um compilador capaz de otimizar o espaço. Do compilador Delta para Modula-2 foi escolhido porque ele é relativamente simples para produção de uma excelente codificação e foi escrito por uma pessoa em poucos meses.

12.1 EQN, UM PRÉ-PROCESSADOR PARA COMPOSIÇÃO DE TIPOS MATEMÁTICOS

O conjunto de possíveis entradas para um interpretador de programas de computador pode ser visto como uma pequena linguagem. A estrutura desses inputs pode ser descrita por uma gramática e uma tradução dirigida pela sintaxe pode ser usada para especificar precisamente o que o programa faz. A tecnologia de compiladores pode então ser aplicada para implementar o programa.

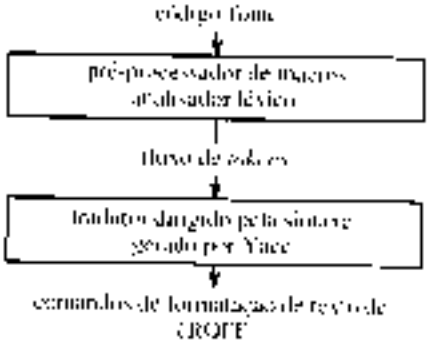


Fig. 12.1. Implementação do EQN

Um dos primeiros compiladores para pequenas linguagens foi o incluído no programa UNIX do EQN, por Kernighan e Cherry [1975]. Como descrito anteriormente na Seção 1.2, EQN formata expressões matemáticas para o formatador de texto TROFF, produzindo uma saída da forma *f*.

A implementação do EQN é descrita na Fig. 12.1. O pré-processamento de macros é descrito na Seção 1.4 e a maioria dessas macros são feitas sobre simplesmente. O fluxo de tokens após a análise léxica é traduzido dirigido pela análise sintática, pelos comandos de formatação de texto. O tradutor é construído usando o gerador de analisadores sintáticos Yacc descrito na Seção 1.9.

O enfoque de tradução usado para EQN combina a linguagem e aplicações ao programa de compiladores para construção de tradutores possui vários benefícios notados pelos autores:

1. *Facilidade de implementação.* "Construções de sistema operacional suficiente para executar os exemplos significativos vieram rapidamente, talvez por uma ordem de magnitude."

2. *Evolução da linguagem.* "Uma definição dirigida pela sintaxe facilita as mudanças na linguagem à medida que ela amadurece. Durante meus EQN evoluiu em resposta às necessidades dos usuários."

Os autores concluem observando que "definir uma linguagem e escrever um compilador para ela mostrou-se um compilador de compiladores poderoso e uma maneira sensível de se trabalhar engenharia".

12.2 COMPILADORES PARA PASCAL

O projeto de Pascal é claramente influenciado pelo projeto compilador para computadores "fortemente dependentes" (ver referência Wirth [1971]). Em consequência, nos interessa examinarmos a estrutura de compiladores para a linguagem escrita por Wirth e seus colegas. O primeiro (Wirth [1971]) e o segundo compiladores (Ammann, in [1981, 1977]) geravam código-objeto de máquina para os equipamentos máquina da série CDC 6000. Experimentos de portabilidade com o segundo compilador levaram ao compilador Pascal-P, que gera instruções chamadas P-code para uma máquina de pilha abstrata (Nori *et al.* [1981]).

Cada um dos compiladores é produto de uma passagem, organizado em torno de uma análise sintática descendente recursiva, como a interface de conjunto "menos" de Capítulo 2. Wirth [1971] observa que "tornou-se relativamente fácil modelar uma linguagem de

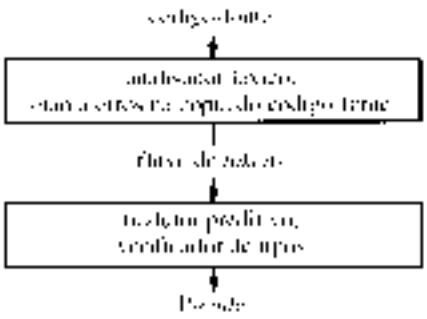


Fig. 12.2. Compilador Pascal P

acordo com as restrições de portação descritas na seção anterior. A organização do compilador Pascal P é mostrada na Fig. 12.2.

As operações básicas da máquina abstrata de pilha usadas pelo compilador Pascal P refletem as necessidades de Pascal. A memória da máquina é organizada em quatro áreas:

1. código para procedimentos;
2. constantes;
3. uma pilha para os registros de ativação;
4. uma área para os dados alocados através da operação `new`.

Todos os procedimentos podem ser armazenados em Pascal-P, e registro de ativação para um procedimento contém todos os elos de acesso e ponteiros de controle. Uma variável do procedimento é traduzida numa instrução de "mãe em pilha" para a máquina abstrata com o elo de acesso apropriado como argumento. O código para o procedimento se refere a menos a posições fixas locais ao seu registro de ativação, apontado por um de uma das cinco unidades nos registros de ativação. A memória para as variáveis locais é alocada através de um ponteiro mantido em um número de elos de acesso e acessada através deles, como descrito na Seção 7.4. O ponteiro compilador usa um display para o acesso eficiente aos não-locais.

Appelbaum [1981] descreveu as seguintes conclusões a partir da experiência de escrever o segundo compilador Pascal usado na Bell Laboratories. O compilador foi escrito em uma versão inicial de Pascal e gerava instruções de uma máquina de pilha de endereço zero. O código para o compilador foi portado para o PDP-11 e para o VAX-11/780. Ele foi escrito para ser compilado em memória secundária. Por outro lado, a organização em uma passagem impossibilitou certas otimizações sobre a qualidade do código gerado, partes de exigências relativamente altas de memória.

12.3 OS COMPILADORES C

C é uma linguagem de programação de propósito geral projetada por Dennis M. Ritchie e é usada como a linguagem primária de programação do sistema operacional UNIX (Ritchie e Thompson [1974]). O próprio UNIX e os utilitários C são usados em uma ampla variedade de máquinas que vão de microprocessadores a grandes mainframes. Instalando-se apenas um compilador C. Esta seção descreve brevemente a estrutura geral dos compiladores para o PDP-11 por Ritchie [1979a] e a família PCC de compiladores C portáveis por Johnson [1979a]. Dois quartos do código de PCC são independentes da máquina alvo. Todos esses compiladores são essencialmente de duas passagens, e o compilador do PDP-11 possui uma terceira passagem opcional que realiza a otimização da

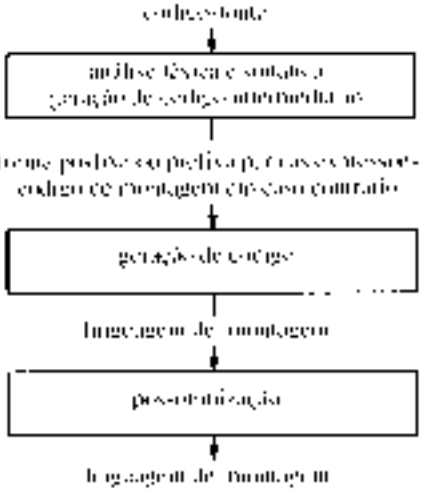


Fig. 12.3. Estrutura de passagens dos compiladores C

saída em linguagem de montagem, como indicado na Fig. 12.3. Esses tipos de otimização são descritos em detalhes no Capítulo 10.

A passagem 1 de cada compilador realiza a análise léxica e sintática e gera código intermediário. O compilador PDP-11 usa a descendência recursiva para a análise sintática, gerando código em forma posfixa para as quais a precedência de operadores é usada. O código intermediário consiste em uma notação posfixa para as expressões e código de montagem para as estruturas de fluxo de controle. O compilador VAX-11/780 usa um analisador sintático LALR(1) gerado por Yacc. Seu código intermediário consiste em uma notação posfixa para expressões e código de montagem para as estruturas de controle. Em cada caso, a alocação de memória para as variáveis locais é feita durante a passagem 1 e dessas locações relativas podem ser referenciados através dos deslocamentos dentro do registro de ativação.

Dentro da interface de montagem, as expressões são representadas por árvores sintáticas. No compilador do PDP-11, a geração de código é implementada através de uma camada da máquina virtual, mesmo de uma estratégia similar à do algoritmo de emulação da Seção 8.1. As modificações naquele algoritmo levam em conta o fato de assegurar que pares de registradores sejam separados para as operações que deles necessitam e também para tirar vantagem dos operandos que possuem constantes.

Johnson [1978] revê a influência da teoria sobre o PCC. Em ambos, PCC e PCC 2 (que veio subsequentemente), o código para as expressões é gerado através da reescrita de árvores. O gerador de código PCC examina uma representação da linguagem de montagem cada vez, procurando especificamente por sub-árvores que possam ser operadas sem armazenamentos usando os registradores dos próprios. Os resultados, computados como na Seção 9.10, identificam as subexpressões a serem computadas e armazenadas em temporários. O código para avaliar e armazenar os valores temporários também é gerado pelo compilador à medida que as subárvores são selecionadas. A reescrita é mais evidente em PCC2, que gera um descritor estruturado e um algoritmo de programação dinâmica da Seção 9.11.

Johnson e Ritchie [1981] descrevem a influência da máquina alvo sobre o projeto dos registros de ativação e da sequência de chamada e retorno de procedimentos. A função de biblioteca `printf` pode ser um exemplo de chamada com algumas máquinas e dominada pela necessidade de permitir listas de tamanho variável para os argumentos.

12.4 OS COMPILADORES FORTRAN H

O compilador original de Fortran H, escrito por Lowry e Medlock [1969], era um abrangente e muito eficiente compilador que

Durante, construído inteiramente em Assembler, os métodos que não exploram amplamente aqueles descritos neste livro. Várias tentativas foram feitas para desenvolver versões mais sofisticadas do compilador, e uma versão estendida do compilador foi desenvolvida para o IBM/370 e uma versão "melhorada" foi desenvolvida por Scarborough e Kolsky [1980]. Fortran H oferece ao usuário a escolha de realizar ou não otimização, alocação de registradores somente ou otimização completa. O desempenho do compilador no caso da otimização completa em cobol está aparecendo na Fig. 12.4.

O texto fonte é analisado em quatro passagens. As duas primeiras realizam a análise léxica e a sintática, produzindo quádruplas. A passagem seguinte incorpora a otimização de código e de registradores e a passagem final gera código objeto a partir das quádruplas e das atribuições de registradores.

A fase de análise léxica é um tanto incomum, pois cada vez que uma sílaba básica (uma cadeia de letras ou uma cadeia de "outros operadores") que não seja precedida por um operando for encontrada, ela será tratada como um operando. Deveria ser notado que, em Fortran, exceto na opinião das linguagens, nunca temos dois tokens operandos consecutivos (tais como identificadores ou constantes inteiras). Assim, *tokens* estão sempre separados por, pelo menos, um token de pontuação.

Por exemplo, o comando de atribuição

```
[illegible]
```

será tratado na seguinte sequência de passos

```
"comando de atribuição"
[illegible]
```

A fase da análise léxica distingue entre um parêntese esquerdo que segue uma função e um introduzir uma. Isto é feito através de um subscrito, daqueles cujo valor é 0 se agrupam os operandos. Por exemplo, o símbolo "(" pretende-se que represente um parêntese esquerdo usado como um operador de subscrições. Os parênteses esquerdos que não têm um operador

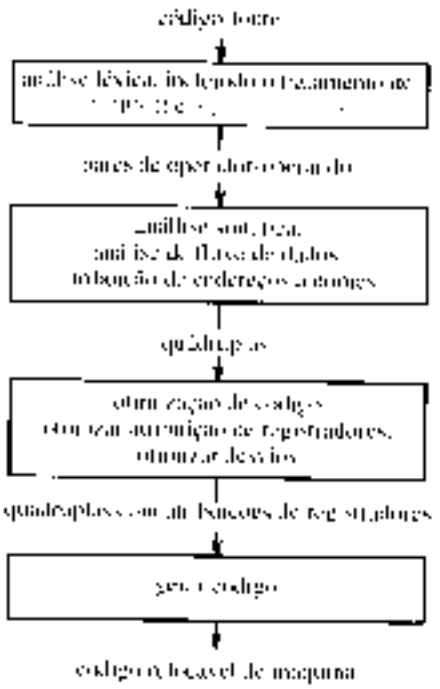


Fig. 12.4. Estrutura do compilador Fortran H

associado à função são reconhecidos posteriormente de maneira [illegible]. É possível testar esta opção, para cada bloco básico, de memória. Para compilações de menor custo associadas as sub-rotinas, é determinar a localização de cada variável [illegible] pelo programa numa dessas áreas de memória estática.

Como Fortran não possui mecanismos de controle de estruturas, como os encontrados em Algol, a análise sintática é bastante direta. Fortran H simplesmente usa uma análise sintática de precedência de operadores para expressões. Algumas otimizações locais muito simples são realizadas durante a geração de quádruplas, por exemplo, multiplicações por potências de 2 são substituídas por operações de deslocamento para a esquerda.

Otimização de Código em Fortran H

Cada subrotina é particionada em blocos básicos e a estrutura de laço é determinada encontrando-se os nodos no grafo de fluxo cujos cabeços dominem suas caudas, como descrito na Seção 10.4. O compilador realiza as seguintes otimizações:

1. *Eliminação de subexpressões comuns.* O compilador procura por subexpressões locais comuns e por expressões que sejam comuns a um bloco B e um dos blocos que B domina. Duas instâncias de subexpressões comuns não são detectadas. Mais ainda, a detecção de subexpressões comuns só leva uma expressão a cada vez, ao invés de se tentar o máximo de cobertura descrito na Seção 10.6. Consequentemente, ao se descartar a versão "melhorada" do compilador, os autores estão supondo que uma cobertura ótima não possível levaria a um nível de melhoria muito pequeno de não.

2. *Movimentação de código.* Os comandos de atribuição são movidos dos laços exceto aqueles que se encontram em [illegible] descrito na Seção 10.7.

3. *Propagação de cópia.* De novo, isto é feito uma vez a cada vez.

4. *Eliminação de variáveis de indução.* Essa otimização é realizada antes de [illegible] que reconhece a otimização uma vez [illegible]. Os passos de [illegible] descrita na Seção 10.7, são feitos [illegible] passagens através do grafo para identificar variáveis de indução que pertençam à família de algumas outra variável de indução.

Através da análise do fluxo de dados, [illegible] um valor constante e os valores correspondentes ao que chamamos [illegible] são armazenados como valores de uso. Entretanto, no compilador original, o limite de tamanho 127 foi colocado nesses valores, e, consequentemente, os grandes programas têm somente as suas variáveis mais frequentemente usadas otimizadas em atribuições. As versões melhoradas [illegible] tais limites.

Otimizações Algébricas

Como a moeda em Fortran é frequentemente usada para computações numéricas, a otimização algébrica é perigosa uma vez que as transformações de expressões podem ter aritmética de computadores tais como overflows de expoente ou perdas de precisão que não estão visíveis se [illegible] uma visão idealizada da semântica algébrica. No entanto, as transformações algébricas envolvendo inteiros são geralmente seguras e a versão melhorada do compilador realiza algumas dessas otimizações somente nos casos de referências a inteiros.

Em geral, uma referência a [illegible] envolve uma cascata de [illegible] que uma expressão da forma [illegible]

[illegible] é computado e os valores exatos das constantes dependendo da localização de [illegible] e das dimensões do arranjo. Se, digamos, [illegible] forem constantes, quer sejam constantes numéricas ou variáveis laço-invariantes, o compilador aplica a lei comutativa e associativa para obter uma expressão da forma [illegible].

Otimização de Registradores

Fortran H divide os registradores em três classes. Esses conjuntos de registradores são usados para a otimização local de registradores, a otimização global de registradores e a otimização de desvios[7]. O número exato de registradores em cada classe pode ser ajustado pelo compilador, dentro de limites.

Os registradores globais são alocados, laço a laço, às variáveis mais frequentemente referenciadas no laço em questão. Uma variável que se qualifica por um registrador em um laço L, mas não no laço que o contém imediatamente L_1, é carregada no pré-cabeçalho de L e armazenada à saída de L.

Os registradores locais são usados dentro de um bloco básico para guardar os resultados de um enunciado até que sejam usados em um ou mais enunciados subsequentes. Somente se não existirem registradores locais suficientes é que um valor temporário é armazenado. O compilador tenta computar novos valores no registrador que guarda um de seus operandos, se aquele operando for subsequentemente eliminado. Na versão melhorada, é feita uma tentativa para reconhecer a situação onde os registradores globais podem ser intercambiados com outros registradores para aumentar o número de vezes que uma operação possa ocorrer no registrador que abrigue um de seus operandos.

A otimização de desvios é um artifício do conjunto de instruções do IBM/370 que paga um prêmio significativo ao desvio somente para localizações que possam ser expressas como o conteúdo de algum registrador mais uma constante no intervalo de 0 a 4095. Por conseguinte, Fortran H aloca alguns registradores para guardar endereços no espaço de código, em intervalos de 4096 bytes, de forma a permitir desvios eficientes em todos os programas, menos os extremamente grandes.

12.5 O COMPILADOR BLISS/11

Esse compilador implementa a linguagem de programação de sistemas Bliss num PDP-11 (Wulf et al. [1975]). Num certo sentido, é um compilador otimizante de um mundo que deixou de existir, aquele em que o espaço de memória estava a prêmio numa escala suficiente para fazer sentido realizar otimizações cujo único propósito era o de reduzir espaço, às vezes às custas de tempo. No entanto, a maioria das otimizações realizadas pelo compilador ganham tempo igualmente, e os descendentes desse compilador estão em uso hoje em dia.

O compilador merece nossa atenção por várias razões. Seu desempenho de otimização é forte e realiza transformações que não são encontradas em lugar nenhum. Sobretudo, foi o pioneiro na abordagem "dirigida pela sintaxe" para a otimização, como discutido na Seção 10.5. Isto é, a linguagem Bliss foi projetada para produzir somente grafos de fluxo redutíveis (não possuía desvios). Por conseguinte, foi possível que a análise de fluxo de dados fosse realizada sobre a árvore gramatical diretamente, ao invés de no grafo de fluxo.

O compilador opera numa única passagem, com um procedimento processado completamente antes do próximo ser lido. Os projetistas vêem o compilador como composto por cinco módulos, como mostrado na Fig. 12.5.

LEXSYNFLO realiza a análise léxica e sintática. Um analisador de descendência recursiva é usado. Na medida em que BLISS não permite comandos de desvio, todos os grafos de fluxo de procedimentos BLISS são redutíveis. De fato, a sintaxe da linguagem nos habilita a construir os grafos de fluxo e determinar os laços e entradas de laços, na medida em que analisamos sintaticamente. LEXSYNFLO assim o faz e também determina as subexpressões comuns e uma variante dos encadeamentos [illegible], tirando vantagem da estrutura dos grafos de fluxo redutíveis. Outra tarefa importante de LEXSYNFLO é a de detectar grupos de expressões similares. Existem candidatas para substituição por uma única subrotina. Note-se que essa substituição faz com que o programa rode mais lentamente, mas pode economizar espaço.

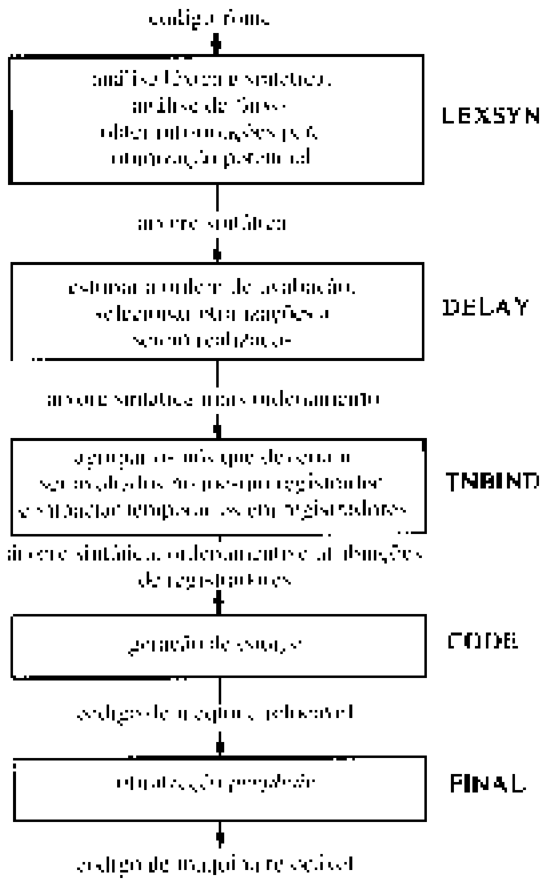


Fig. 12.5 O compilador BLISS/11.

O módulo DELAY examina a árvore sintática para determinar quais instâncias particulares das otimizações usuais, tais como a movimentação de código invariante e a eliminação de subexpressões comuns, estão realmente indicadas a produzir lucro. A ordem de avaliação de expressões é determinada nesse tempo, baseada na estratégia de rotulação da Seção 9.10, modificada para levar em conta os registradores que não estão disponíveis por estarem sendo usados para preservar os valores das subexpressões comuns. As leis algébricas são usadas para determinar se o reordenamento das computações deveria ser feito. As expressões condicionais são avaliadas numericamente ou através do fluxo de controle, como discutido na Seção 8.4, e DELAY decide qual modo é mais barato em cada instância.

TNBIND considera quais nomes temporários deveriam ser atribuídos a registradores. Tanto os registradores quanto as localizações de memória são alocados. A estratégia usada é primeiro agrupar os nós da árvore sintática que deveriam estar associados ao mesmo registrador. Como discutido na Seção 9.6, há vantagem em se avaliar um nó no mesmo registrador que um de seus pais. Em seguida, a vantagem a se ganhar mantendo-se um temporário num registrador é estimada por um cálculo que favorece aqueles que são usados várias vezes sobre um pequeno intervalo. Os registradores são atribuídos até que usados

[7] [illegible] IBM/370 [illegible]

compiladores produzidos pelo gerador de geradores YACC/PCC. Esse compilador é razoavelmente eficiente, com seu predomínio de instruções de atribuição de registradores para cálculos de relações de indicadores.

Esse código é, em seguida, especialmente examinado por UNOP, que realiza a otimização peephole de que já resultem muitas melhorias. As melhorias feitas incluem a eliminação de cópias para testes condicionais ou incondicionais e a complementação das condições, como descrito na Seção 9.9.

As instruções redundantes e outras ineficiências são eliminadas. Essas podem ainda resultar de comandos como os de ENSAIO A, combinação de sequências de códigos simulares nas duas limitações de um desvio e levadas, como consequência, à propagação de constantes. Um número de outras otimizações locais, algumas realmente dependentes da máquina, é tentado. Uma dessas otimizações importantes, que é substituição, sempre que for possível, de instruções de desvio por "instruções" PDP-11, que requerem um palavra, mas estão limitadas em 128 bytes em seu alcance.

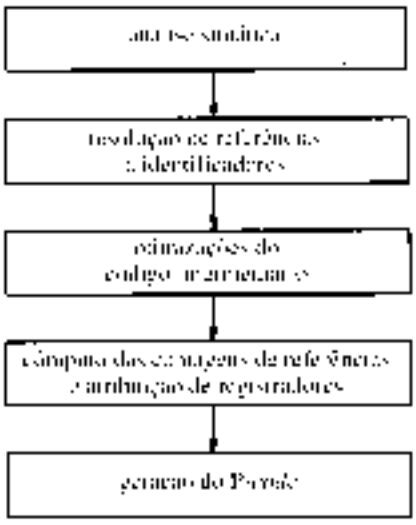

análise sintática
resolução de referências a identificadores
otimizações do código intermediário
computação das exigências da referências e atribuição de registradores
geração do Pascal

Fig. 12.6. Passos sobre o compilador Modula-2

12.6 O COMPILADOR OTIMIZANTE MODULA-2

Esse compilador, descrito em Powell 1984, foi desenvolvido com a finalidade de produzir código de boa qualidade, usando otimizações que providenciam um arcabouço para um pequeno esboço. O autor descreve sua estratégia como "dominada pelas melhores otimizações simples". Tal filosofia pode ser difícil de conciliar com experimentação e modificações rápidas, mas algumas das "melhores otimizações simples" aqui empregadas incluem algumas das decisões de projeto do compilador de Modula-2 não prontamente adequadas para um compilador que providencie máxima otimização. Contudo, a estratégia aqui descrita produz um código excelente com um compilador que pode ser escrito em uns poucos meses por uma pessoa. Os cinco passos do compilador são delineados na Fig. 12.6.

O analisador sintático foi gerado usando o YACC e compõe um produto que vem sofrer as emendas e passagens pelas variáveis de Modula não. Em que ser declaradas antes do uso. Esse é uma tentativa de tornar o compilador compatível com facilidades existentes. O código intermediário é Pascal, para compatibilidade com outros compiladores Pascal-P, que rodam sob Berkeley UNIX, e, por conseguinte, os procedimentos escritos nesses linguagens podem ser incorporados ao produto.

O compilador não realiza análise de fluxo de dados. Através Modula-2, como Bliss, é uma linguagem que pode produzir somente grafos de fluxo reduzíveis, e, dessa forma, a metodologia de Seção 10.5 pode ser aplicada com certa facilidade. De fato, o compilador de Modula-2 serve além do compilador BLISS-11, pois apresenta como vantagem da sintaxe. Os laços são identificados por suas sintaxes, isto é, o compilador procura por sucessões sobre o laço. As expressões invariantes são detectadas pelo fato de que nenhuma de suas variáveis são definidas no laço, sendo essas expressões armazenadas para serem chamadas no mesmo. As únicas variáveis de indução que são detectadas são as que envolvem uma comparação de igualdade de laço for. As outras possíveis variáveis são detectadas quando uma delas ocorre no lado esquerdo de uma atribuição e tenha uma expressão de cada vez, através de sequências de *loads*.

A estratégia de alocação de registradores é similarmente projetada para realizar coisas razoáveis sem ser sofisticada. Em particular, são considerados como candidatos para alocação a um registrador somente

1. os temporários usados quando a avaliação de uma expressão lhes recebem a primeira prioridade;
2. valores de subexpressões comuns;
3. índices e valores limite de laços for;
4. o endereço de A onde a expressão da forma A[I] é
5. variáveis simples locais de tipos inteiros ou ponteiros não preditas ou previamente ocorrente.

É feita uma tentativa para estimar o valor de se manter cada uma das variáveis classes 2 e 5 em um registrador. É assumido que se um enunciado é executado 10 vezes, ele está aninhado em relação aos anteriores. Se o código é determinado que mais de que duas vezes não são considerados elegíveis, outras são baseadas pela ordem de uso estimado e atribuídas a um registrador. Se não estiver disponível, após atribuídos os temporários de expressões e variáveis de índice, [illegible]

APÊNDICE

UM PROJETO DE PROGRAMAÇÃO

A.1 INTRODUÇÃO

Este apêndice sugere exercícios de programação que podem ser usados num laboratório de programação associado a um curso de projeto de compiladores baseado neste livro. Os exercícios consistem na implementação dos construtores básicos de um compilador para um subconjunto de Pascal. O subconjunto é pequeno, mas permite que programas bem comportados sejam escritos. Sendo um subconjunto de uma linguagem de programação existente, possui uma certa utilidade. Os gramáticos dos programas no subconjunto podem ser determinados pela semântica de Pascal (Jensen e Wirth [1975]). Se um compilador Pascal estiver disponível, pode ser usado como referência para verificar o comportamento do compilador escrito. As construções do subconjunto apresentam uma variedade das linguagens de programação, e, dessa forma, exercícios construídos podem ser compilados usando-se uma linguagem diferente, se um compilador Pascal não estiver disponível.

A.2 ESTRUTURA DE PROGRAMA

Um programa consiste em uma sequência de declarações de dados globais, de declarações de procedimentos e funções e de um único comando composto que é o "programa principal". Para os dados globais deve ser alocada memória estática. Os dados locais aos procedimentos e funções terão a memória alocada numa pilha. A recursão é permitida. Os parâmetros são passados por referência. Assume-se que os procedimentos [illegible] serão fornecidos pelo compilador.

A Fig. A.1 mostra um programa exemplo. O nome do programa é [illegible] são os nomes dos arquivos usados [illegible] respectivamente.

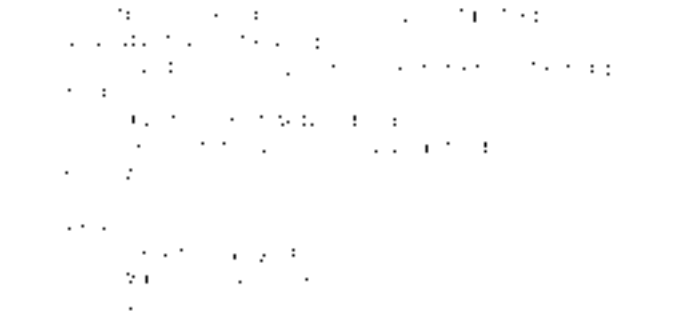


Fig. A.1. Programa exemplo.

A.3 SINTAXE DE UM SUBCONJUNTO DE PASCAL

Listada abaixo está uma gramática LALR(1) para um subconjunto de Pascal. A gramática pode ser modificada para permitir análise sintática de recursão à esquerda como descrito nas Seções 2.4 e 4.3. Uma análise sintática de precedência de operadores pode ser evitada para expressões através da substituição de **operador_relacional**, **operador_aditivo** e **operador_multiplicativo** e das respectivas produções e.

A adição da produção

comando → **if** *expressão* **then** *comando*

cria uma ambiguidade do "if-then-else", que pode ser eliminada conforme discutido na Seção 4.8 ou 5.4 ou 6.4 ou 6.5 ou 6.6 ou 6.7 ou 6.8 ou 6.9 e a análise sintática preditiva também pode ser usada.

Não existe distinção sintática entre uma variável simples e uma chamada de função sem parâmetros. Ambas são geradas pela produção

fator → **id**

Para conseguir a distinção [illegible] estabelece [illegible] correta durante a fase [illegible] pela função [illegible] tiver sido declarada como uma função.

programa →
  program id *(lista_de_identificadores)*
  declarações
  declarações_de_subprogramas
  comando_composto
  .

lista_de_identificadores →
  id
  | *lista_de_identificadores* **,** **id**

declarações →
  declarações **var** *lista_de_identificadores* **:** *tipo*
  | ε

tipo →
  tipo_padrão
  array **[** **num** **..** **num** **]** **of** *tipo_padrão*

tipo_padrão →
  inteiro
  | **real**

```
declaração_de_subprograma →
        cabeçalho_de_subprograma declarações_locais corpo_de_subprograma
        ;

declaração_de_subprograma →
        cabeçalho_de_subprograma
        declarações
        encaminhamento_externo

cabeçalho_de_subprograma →
        function id argumentos tipo_resultado ;
        | procedure id argumentos ;

argumentos →
        ( lista_de_parâmetros )
        | ε

lista_de_parâmetros →
        lista_de_identificadores : tipo
        | lista_de_parâmetros ; lista_de_identificadores : tipo

corpo_de_subprograma →
        begin
        sentenças_opcionais
        end

sentenças_opcionais →
        lista_de_sentenças
        | ε
lista_de_sentenças →
        sentença
        | lista_de_sentenças ; sentença

sentença →
        variável operador-de-atribuição expressão
        | chamada_de_procedimento
        | sentença_composta
        | if expressão then sentença else sentença
        | while expressão do sentença

variável →
        id
        | id [ expressão ]

chamada_de_procedimento →
        id
        | id ( lista_de_parâmetros )

lista_de_expressões →
        expressão
        | lista_de_expressões , expressão

expressão →
        expressão_simples
        | expressão_simples operador_relacional expressão_simples

expressão_simples →
        termo
        | expressão_simples operador_aditivo termo

termo →
        fator
        | termo operador_multiplicativo fator

fator →
        id
        | id ( lista_de_parâmetros )
        | num
        | ( expressão )
        | not fator

sinal →
        + | -
```

A.4 CONVENÇÕES LÉXICAS

A notação para a especificação de tokens é a mesma descrita na Seção 3.3.

1. Os comentários são envolvidos por { e }, não podendo conter } dentro deles. Os comentários podem aparecer após qualquer token.
2. Os espaços entre tokens são ignorados, exceto quando delimitam tokens. Os tokens também podem ser delimitados por espaços em branco, isto é, espaços, tabulações e mudanças de linha, exceto no interior de palavras-chave, operadores compostos e identificadores.
3. O token id são identificadores constituídos de uma letra seguida por zero ou mais letras ou dígitos.

```
letra  →  a | b | ... | z | A | B | ... | Z
dígito →  0 | 1 | ... | 9
id     →  letra | id letra | id dígito
```

O implementador pode decidir se quer ou não distinguir entre maiúsculas e minúsculas no reconhecimento de identificadores.

4. Os números inteiros são constituídos de uma ou mais ocorrências de dígitos. Números reais são constituídos de uma parte inteira, um ponto decimal, uma parte fracionária e um expoente opcional. A parte fracionária pode ser zero ou mais dígitos, mesmo que seja vazia. Um ponto decimal não precisa ser seguido de dígitos se não houver parte fracionária. Um expoente é constituído de um e ou E, um sinal opcional e uma ou mais ocorrências de dígitos. Se houver um expoente, a parte fracionária pode ser vazia. Se não houver expoente, a parte fracionária não pode ser vazia. Se houver parte fracionária, a parte inteira pode ser vazia. Se não houver parte fracionária, a parte inteira não pode ser vazia. Exemplos válidos de números reais são 1., 1.0, .1, 1.1, 1.e1, 1.1E-1, 1.1e+1. Exemplos inválidos são . e .e1. Os números inteiros não podem ter zeros à esquerda, exceto o próprio zero. Exemplos válidos de números inteiros são 0, 1, 10 e 101. Exemplos inválidos são 00, 01 e 010. Números inteiros não podem ter expoentes. Exemplos válidos de números reais são 0.0, 0.1, 1.0, 1.1, 1.1e1, 1.1E-1, 1.1e+1, 0.1e1, 0.1E-1, 0.1e+1, 1.e1, 1.E-1, 1.e+1, .1e1, .1E-1 e .1e+1. Exemplos inválidos são 00.0, 01.0, 010.0, 1.1e0, 1.1e-0, 1.1e+0, 1.1e01, 1.1e-01 e 1.1e+01.

```
dígitos →  dígito dígito*
fração opcional →  ε | . dígitos
expoente opcional →  ε | ( e | E ) ( + | - | ε ) dígitos
num →  dígitos fração opcional expoente opcional
```

5. As palavras-chave são sempre delimitadas e aparecem em negrito na gramática.
6. Os operadores relacionais operador_relacional são =, <>, <, <=, >= e >. Note que = é igual a, não atribuição.
7. Os operadores aditivos operador_aditivo são +, -, or e +.
8. Os operadores multiplicativos operador_multiplicativo são *, /, div, mod, and e *.
9. O lexema para o token operador_de_atribuição é :=.

A.5 EXERCÍCIOS SUGERIDOS

Um exercício de programação adequado para um curso de compiladores é escrever um interpretador para a linguagem de uma calculadora ou para um subconjunto similar de outra linguagem de alto nível. O projeto envolve a tradução do programa-fonte em uma representação intermediária (tal como pós-fixa ou código de máquina de pilha) e a interpretação dessa representação. Temos sugerido também para os estudantes que escrevam um compilador para traduzir programas-fonte em código de máquina. A ordem e a forma em que as rotinas são executadas no compilador porque é conveniente ter um interpretador operativo para depurar os outros componentes do compilador.

1. *Procurar ou inserir um token na tabela de símbolos.* Deve ser responsável por organizar a tabela de símbolos. Para cada token que seja colocado na tabela, armazenar informações a respeito de nomes, mais tarde. Deve ser construída uma rotina de pesquisa da tabela de símbolos por nome. Escrever rotinas para:
   i. Pesquisar a tabela de símbolos procurando por um token cadeia e retornar uma nova entrada para o token se o token não estiver na tabela. Deve retornar uma indicação para que seja possível determinar se o token é novo ou não.
   ii. Remover da tabela de símbolos todos os nomes locais a um dado procedimento.
2. *Escrever um analisador para expressões.* O compilador ou calculadora pode ser derivado escrevendo-se primeiro uma rotina para reconhecer expressões e, em seguida, aprimorá-la de modo a incluir outras construções com base nas operações da linguagem. Incluir gradativamente operações lógicas, seus códigos de operação e rotinas de geração de código. Incluir também a necessidade de "quádruplas" para a conversão de inteiro para real, para marcar o início e o final de procedimentos e para a transmissão de parâmetros e chamadas de procedimentos.

É também necessário, a esse tempo, projetar a sequência de chamadas e a organização em tempo de execução para os programas que serão interpretados. A organização simples de pilha discutida na Seção 6.3 é adequada à linguagem exemplo, porque nenhuma declaração aninhada ou procedimento é permitido. Os nomes não declarados como globais são declarados através de todo o programa ou locais a um procedimento simples.

Para uma questão de simplicidade, outra linguagem de alto nível pode ser usada em lugar do interpretador. Outra opção é projetar e escrever um compilador para uma linguagem de alto nível como Pascal ou Pascal. Você deve compilar e rodar o código numa sequência de exemplos. Depois, em um projeto mais complexo, as palavras-chave reservadas, a tabela de símbolos, Pascal ou Pascal são usadas para ser uma subrotina chamada pelo programa principal, que é compilado por você sob demanda.

4. *Análise semântica e geração de código intermediário.* Selecionar as rotinas semânticas para gerar as quádruplas. Você também necessitará ser modificado em alguns locais para formar a tradução dirigida pela sintaxe final. Consultar as Seções 5.2 e 5.3 para os exemplos sobre como modificar a gramática de forma útil. Realizar as ações semânticas a esse tempo, como descrito anteriormente para os testes quando necessário.

5. *Detecção e recuperação de erros.* Se um gerador de analisadores sintáticos (YACC) estiver disponível, escrever simplificar a tarefa consideravelmente. Se um gerador de analisadores sintáticos que trate gramáticas ambíguas como YACC estiver disponível, as não-determinações que deixam expressões podem ser computadas. Sobretudo a ambiguidade do "else-casa" pode ser resolvida empilhando-se o símbolo que um conflito de empilha/reduz ocorrer.

6. *Inclusão de rotinas de tratamento de erros.* Estar preparado para se recuperar dos erros léxicos e sintáticos. Imprimir diagnósticos de erro para os erros léxicos, sintáticos e semânticos.

7. *Testagem.* O programa da Fig. A.1 pode ser útil como uma simples rotina de teste. Um outro programa de teste pode ser baseado no programa da Fig. 7.1. O código para a função [illegible] gramáticas arbitrárias do programa Compilador da Fig. III.2 Pode ser compilado através de um gerador de perfis de computação, se um estiver disponível. Determinar as rotinas nas quais a maior parte do tempo é gasto. Quais rotinas devem ser modificadas para se aumentar a velocidade do seu compilador?

A.6 EVOLUÇÃO DO INTERPRETADOR

Uma abordagem alternativa para se escrever um interpretador para linguagens é começar implementando uma calculadora de bolso, depois um interpretador para expressões booleanas, depois variáveis, depois a linguagem, até que um interpretador para a linguagem completa seja obtido. Uma abordagem similar é usada em Kernighan e Pike [1984]. Uma ordem proposta para se adicionar as construções é:

1. *Escrever expressões, depois declarações.* Usando uma análise sintática recursiva descendente, como no Capítulo 2, ou um gerador de analisadores sintáticos, começar a escrever o ambiente de programação, escrevendo um tradutor de expressões infixadas simples para a notação posfixa.
2. *Adicionar variáveis globais e booleanas.* Permitir que palavras-chave identificadores e números ignorem tudo o que for necessário. Reorganizar o tradutor de forma a que produza um código para uma máquina de pilha ou quádruplas.
3. *Escrever um interpretador para a representação intermediária.* Como discutido na Seção 5.7, uma linguagem de alto nível pode ser usada como base do interpretador. Para o momento, o interpretador necessita somente suportar operações aritméticas, atribuições e entrada e saída. Expandir a linguagem, permitindo declarações de [illegible] variáveis globais, atribuições e chamadas aos procedimentos [illegible] e [illegible]. Essas construções permitem que o interpretador seja testado.
4. *Adicionar procedimentos.* Um programa na linguagem como se segue em um pequeno ponto por com declarações de subprogramas. Testar ambos, tradutor e interpretador.
5. *Adicionar procedimentos e funções.* A tabela de símbolos precisa agora permitir que os escopos das identificadores sejam tratados através dos procedimentos. Projetar uma sequência de chamada. Depurar o interpretador para suportar a sequência de chamada.

A.7 EXTENSÕES

Existe um número de ligações que podem ser adicionadas à linguagem e também a complexidade da compilação. Dentre essas estão:

1. arrays unidimensionais
2. strings
3. estrutura de bloco
4. estrutura de registro

Se o tempo permitir, adicionar uma ou mais dessas extensões ao seu compilador.

No original: seguintes [illegible] ... [illegible]

BIBLIOGRAFIA

ABEL, N. E. AND J. R. BELL [1972]. "Global optimization in compilers," *Proc. First USA-Japan Computer Conf.*, AFIPS Press, Montvale, N.J.

ABELSON, H. AND G. J. SUSSMAN [1985]. *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, Mass.

ADRION, W. R., M. A. BRANSTAD, AND J. C. CHERNIAVSKY [1982]. "Validation, verification, and testing of computer software," *Computing Surveys* **14**:2, 159-192.

AHO, A. V. [1980]. "Pattern matching in strings," in Book [1980], pp. 325-347.

AHO, A. V. AND M. J. CORASICK [1975]. "Efficient string matching: an aid to bibliographic search," *Comm. ACM* **18**:6, 333-340.

AHO, A. V. AND M. GANAPATHI [1985]. "Efficient tree pattern matching: an aid to code generation," *Twelfth Annual ACM Symposium on Principles of Programming Languages*, 334-340.

AHO, A. V., J. E. HOPCROFT, AND J. D. ULLMAN [1974]. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass.

AHO, A. V., J. E. HOPCROFT, AND J. D. ULLMAN [1983]. *Data Structures and Algorithms*, Addison-Wesley, Reading, Mass.

AHO, A. V. AND S. C. JOHNSON [1974]. "LR parsing," *Computing Surveys* **6**:2, 99-124.

AHO, A. V. AND S. C. JOHNSON [1976]. "Optimal code generation for expression trees," *J. ACM* **23**:3, 488-501.

AHO, A. V., S. C. JOHNSON, AND J. D. ULLMAN [1975]. "Deterministic parsing of ambiguous grammars," *Comm. ACM* **18**:8, 441-452.

AHO, A. V., S. C. JOHNSON, AND J. D. ULLMAN [1977a]. "Code generation for expressions with common subexpressions," *J. ACM* **24**:1, 146-160.

AHO, A. V., S. C. JOHNSON, AND J. D. ULLMAN [1977b]. "Code generation for machines with multiregister operations," *Fourth ACM Symposium on Principles of Programming Languages*, 21-28.

AHO, A. V., B. W. KERNIGHAN, AND P. J. WEINBERGER [1979]. "Awk — a pattern scanning and processing language," *Software—Practice and Experience* **9**:4, 267-280.

AHO, A. V. AND T. G. PETERSON [1972]. "A minimum distance error-correcting parser for context-free languages," *SIAM J. Computing* **1**:4, 305-312.

AHO, A. V. AND R. SETHI [1977]. "How hard is compiler code generation?" *Lecture Notes in Computer Science* **52**, Springer-Verlag, Berlin, 1-15.

AHO, A. V. AND J. D. ULLMAN [1972a]. "Optimization of straight line code," *SIAM J. Computing* **1**:1, 1-19.

AHO, A. V. AND J. D. ULLMAN [1972b]. *The Theory of Parsing, Translation and Compiling*, Vol. I: *Parsing*, Prentice-Hall, Englewood Cliffs, N.J.

AHO, A. V. AND J. D. ULLMAN [1973a]. *The Theory of Parsing, Translation and Compiling*, Vol. II: *Compiling*, Prentice-Hall, Englewood Cliffs, N.J.

AHO, A. V. AND J. D. ULLMAN [1973b]. "A technique for speeding up LR(k) parsers," *SIAM J. Computing* **2**:2, 106-127.

AHO, A. V. AND J. D. ULLMAN [1977]. *Principles of Compiler Design*, Addison-Wesley, Reading, Mass.

AIGRAIN, P., S. L. GRAHAM, R. R. HENRY, M. K. MCKUSICK, AND E. PELEGRI-LLOPART [1984]. "Experience with a Graham-Glanville style code generator," *ACM SIGPLAN Notices* **19**:6, 13-24.

ALLEN, F. E. [1969]. "Program optimization," *Annual Review in Automatic Programming* **5**, 239-307.

ALLEN, F. E. [1970]. "Control flow analysis," *ACM SIGPLAN Notices* **5**:7, 1-19.

ALLEN, F. E. [1974]. "Interprocedural data flow analysis," *Information Processing 74*, North-Holland, Amsterdam, 398-402.

ALLEN, F. E. [1975]. "Bibliography on program optimization," RC-5767, IBM T. J. Watson Research Center, Yorktown Heights, N.Y.

ALLEN, F. E., J. L. CARTER, J. FABRI, J. FERRANTE, W. H. HARRISON, P. G. LOEWNER, AND L. H. TREVILLYAN [1980]. "The experimental compiling system," *IBM J. Research and Development* **24**:6, 695-715.

ALLEN, F. E. AND J. COCKE [1972]. "A catalogue of optimizing transformations," in Rustin [1972], pp. 1-30.

ALLEN, F. E. AND J. COCKE [1976]. "A program data flow analysis procedure," *Comm. ACM* **19**:3, 137-147.

ALLEN, F. E., J. COCKE, AND K. KENNEDY [1981]. "Reduction of operator strength," in Muchnick and Jones [1981], pp. 79-101.

AMMANN, U. [1977]. "On code generation in a Pascal compiler," *Software—Practice and Experience* **7**:3, 391-423.

AMMANN, U. [1981]. "The Zurich implementation," in Barron [1981], pp. 63-82.

ANDERSON, J. P. [1964]. "A note on some compiling algorithms," *Comm. ACM* **7**:3, 149-150.

ANDERSON, T., J. EVE, AND J. J. HORNING [1973]. "Efficient LR(1) parsers," *Acta Informatica* **2**:1, 12-39.

ANKLAM, P., D. CUTLER, R. HEINEN, JR., AND M. D. MACLAREN [1982]. *Engineering a Compiler*, Digital Press, Bedford, Mass.

ARDEN, B. W., B. A. GALLER, AND R. M. GRAHAM [1961]. "An algorithm for equivalence declarations," *Comm. ACM* **4**:7, 310-314.

ARDEN, BRUCE W., AND R. C. HAINES [1969], "An overview of the PL/6 compiler," *AFIPS Conference Proceedings, Spring Joint Computer Conference* **34**, 189–196.

BAKER, BRENDA S. [1979], "An algorithm for structuring flowgraphs," *JACM* **24**:1, 98–120.

BARNES, JOHN G. P. [1974], "An overview of Euclid," *SIGPLAN Notices* **9**:5, 60–83.

BARNETT, MICHAEL P., AND CHRISTOPH W. GRUND [1994], "Global data flow analysis of explicitly parallel programs," *Proceedings of the 1994 International Conference on Parallel Processing*, 2, 172–179.

BARRON, D. W. [1968], *Recursive Techniques in Programming*, Macdonald, London.

BAUER, FRIEDRICH L., AND HANS WÖSSNER [1972], "The 'Plankalkül' of Konrad Zuse: A forerunner of today's programming languages," *CACM* **15**:7, 678–685.

BEATTY, JOHN C. [1972], "An axiomatic approach to code optimization for expressions," *JACM* **19**:4, 714–724. Errata in **20**:3, 177, p. 160 and 938.

BEATTY, JOHN C. [1974], "Register assignment algorithm for generation of highly optimized object code," *IBM J. Research and Development* **18**:2, 20–39.

BELADY, L. A., [1966], "A study of replacement algorithms for a virtual-storage computer," *IBM Systems J.* **5**:2, 78–101.

BENTLEY, J. L. [1982], *Writing Efficient Programs*, Prentice-Hall, Englewood Cliffs, N.J.

BESS, C. J. H., W. S. CLEVELAND, AND R. SIDAK [1985], "Statistical analysis of hash functions," manuscript, AT&T Bell Laboratories, Murray Hill, N.J.

BIRMAN, A., AND J. D. ULLMAN [1973], "Parsing algorithms with backtrack," *Information and Control* **23**:1, 1–34.

BIRTWISTLE, G. A. [1976], "Semantic evaluation from left to right," *CACM* **19**:2, 95–97.

BOCHMANN, G. V., AND P. WARD [1978], "Compiler writing system for attribute grammars," *Computer J.* **21**:2, 144–148.

BOCH, K. V. [1980], *Formal Language Theory*, Academic Press, New York.

BOYER, R. S., AND J. S. MOORE [1977], "A fast string searching algorithm," *CACM* **20**:10, 762–772.

BRAND, MICHEL, PAUL P. CHEVALIER, ET AL. [1976], "Implementation, Proof and Application of $A_{pl}650$," Lecture Notes in Computer Science, 38, Springer-Verlag, Berlin.

BRATLEY, H. [1971], "An alternate form of the Lyon Algorithm," *CACM* **14**:3, 145.

BROOKES, R. A., AND D. MORRIS [1972], "A general bus-driven program for phrase structure languages," *CACM* **15**:1, 1–11.

BROOKS, F. P., JR. [1975], *The Mythical Man-Month*, Addison-Wesley, Reading, Mass.

BRUNO, JOHN L. [1974], *Deterministic Sequencing Problems*, Ph.D. Thesis, TR 5-71, Harvard University, Cambridge, Mass.

BRUNO, J., AND T. LASSAGNE [1975], "The generation of optimal code for stack machines," *J. ACM* **22**:3, 382–396.

BRUNO, J., AND R. SETHI [1976], "Code generation for one-register machines," *J. ACM* **23**:3, 502–510.

BURSTALL, R. M., J. B. MACQUEEN, AND D. T. SANNELLA [1980], "Hope: an experimental applicative language," *1980 LISP Conference*, 136–143, Redwood Estates, Calif., 1980.

BUSAM, V. A., AND D. E. ENGLUND [1969], "Optimization of expressions in Fortran," *CACM* **12**:12, 666–674.

CAMPBELL, J. [1981], "Basic polystyle types," *Computing Science Technical Report* **92**, AT&T Bell Laboratories, Murray Hill, N.J.

CARLSON, J. K. [1982], *Introduction to Pascal*, rev. ed., CMU Research Computer Center, Ann Arbor, Michigan.

CARLSSON, R. [1985], "Types as intervals," *Journal of the Association for Computing Machinery*, Conference Proceedings, Computers **22**, 46.

CATTELL, R. G. G. [1980], "Automatic derivation of code generators from machine descriptions," *TOPLAS* **2**:2, 173–190.

CHAITIN, G. J. [1982], "Register allocation and spilling via graph coloring," *ACM SIGPLAN '82 Notices* **17**:6, 201–207.

CHAITIN, G. J., M. A. AUSLANDER, A. K. CHANDRA, J. COCKE, M. E. HOPKINS, AND P. W. MARKSTEIN [1981], "Register allocation via coloring," *Computer Languages* **6**, 47–57.

CHERNIAVSKY, J. C., P. H. H. SANDERSON, AND J. KOCHANI [1976], "On the equivalence of RBF flowgraphs and reducible flowgraphs," *Proc. 1976 Conf. Information Sciences and Systems*, Johns Hopkins Univ., 423–429.

CHOW, F. C. [1983], "Writing pools," *IEEE Computer Society International Conference COMPCON*, 100–104.

CHOMSKY, N. [1956], "Three models for the description of language," *IRE Transactions on Information Theory* **IT-2**:3, 113–124.

CHOW, F. [1983], *A Portable Machine Independent Global Optimizer*, Ph.D. Thesis, Computer Systems Lab., Stanford Univ., Stanford, Calif.

CHOW, F., AND J. L. HENNESSY [1984], "Register allocation by priority-based coloring," *ACM SIGPLAN Notices* **19**:6, 222–232.

CHURCH, A. [1941], *The Calculi of Lambda-Conversion*, Annals of Math. Studies No. 6, Princeton University Press, Princeton, N.J.

CHURCH, A. [1956], *Introduction to Mathematical Logic*, Vol. 1, Princeton University Press, Princeton, N.J.

CLARK, J. [1979], "A Preliminary Report on Euclid Type Checking," *SIGPLAN Notices* **14**:1, 16–29.

CLARK, J. [1981], "Global Data Accesses and Expression Optimization," *SIGPLAN Notices* **16**:7, 29–31.

CLARK, J., AND K. KENNEDY [1979], "Flow Graph Decomposition Algorithms," *Computers and Mathematics with Applications* **5**:2, 145–164.

Cocke, J., and K. Kennedy [1977]. "An algorithm for reduction of operator strength." *Comm. ACM* **20**(11), 850–856.

Cocke, J., and J. Marcuello [1969]. "Measurement of code improvement algorithms." *Information Processing 68*, 221–228.

Cocke, J., and P. Markstein [1980]. "Measurement of program improvement algorithms." *Proc. 2nd Int. Workshop on Lisp Conf.*, Stanford University, 1–6.

Cocke, J., and J. T. Schwartz [1970]. *Programming Languages and Their Compilers: Preliminary Notes, Second Revised Version*. Courant Institute of Mathematical Sciences, New York.

Coffman, E. G., Jr., and R. Sethi [1983]. "Instruction sets for evaluating arithmetic expressions." *J. ACM* **30**(3), 457–478.

Colmerauer, A. [1970]. "Total precedence relations." *J. ACM* **17**(1), 14–30.

Conway, M. E. [1963]. "Design of a separable transition-diagram compiler." *Comm. ACM* **6**(7), 396–408.

Conway, R. W., and W. L. Maxwell [1961]. "CORC—the Cornell computing language." *Comm. ACM* **4**(6), 263–264.

Conway, R. W., and T. R. Wilcox [1973]. "Design and implementation of a diagnostic compiler for PL/C." *Comm. ACM* **16**(3), 169–179.

Conway, R. W., and T. R. Wilcox [1977]. "A deterministic LR parser generator." *Proc. IEEE Computer Society Conference*, 305–309.

Cook, S. A. [1971]. "Linear time simulation of deterministic two-way pushdown automata." *Information Processing 71*, 75–80.

Cook, S. A., and S. O. Aanderaa [1969]. "On the minimum computation time of functions." *Trans. Amer. Math. Soc.* **142**, 291–314.

Cormack, G. V. [1981]. "An algorithm for the selection of overlay regions." *Software—Practice and Experience* **11**(11), 1153–1172.

Cormack, G. V., R. N. S. Horspool, and M. Kaiserswerth [1985]. "Practical perfect hashing." *Computer J.* **28**(1), 54–58.

Cornelius, R. [1981]. "Automatic program analysis." *Software—Practice and Experience* **11**(11), 1083–1092.

Cousot, P. [1981]. "Semantic foundations of program analysis." In Muchnick and Jones [1981], pp. 303–342.

Cousot, P., and R. Cousot [1977]. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints." *Conf. Rec. 4th ACM Symposium on Principles of Programming Languages*, 238–252.

Curry, H. B., and R. Feys [1974]. *Combinatory Logic*, Vol. 1. North-Holland, Amsterdam.

Dahl, O.-J. [1966]. *An Approximative Description of Standard Simula*. Norwegian Computing Center, Oslo.

Davidson, J. W., and C. W. Fraser [1980]. "The design and application of a retargetable peephole optimizer." *TOPLAS* **2**(2), 191–202. Errata **3**(1), 66–67.

Davidson, J. W., and C. W. Fraser [1984a]. "Automatic generation of peephole optimizations." *SIGPLAN Notices* **19**(6), 111–116.

Davidson, J. W., and C. W. Fraser [1984b]. "Code selection through object code optimization." *TOPLAS* **6**(4), 505–526.

DeRemer, F. [1969]. *Practical Translators for LR(k) Languages*. Ph.D. Thesis, MIT, Cambridge, Mass.

DeRemer, F. [1971]. "Simple LR(k) grammars." *Comm. ACM* **14**(7), 453–460.

DeRemer, F., and T. Pennello [1982]. "Efficient computation of LALR(1) look-ahead sets." *TOPLAS* **4**(4), 615–649.

Dershowitz, A. J. [1975]. "Elimination of chain productions and merging of nonterminal symbols in LR(k) grammars." *J. Computer Languages* **1**(2), 105–119.

Deutsch, P., R. Dybvig, and J. Hennessy [1984]. "Optimization of power tables for portable compilers." *TOPLAS* **6**(4), 547–572.

Dewar, R. B. K., M. Jacobs, and B. Lorber [1984]. "Dispelling spelling errors: tools for detecting misspellings." *Software—Practice and Experience* **14**(7), 675–687.

Dijkstra, E. [1984]. "Two undecidable problems of finite-state automata." In Kahn, MacQueen, and Plotkin [1984], pp. 213–214.

Dijkstra, E. W. [1960]. "Recursive programming." *Numerische Math.* **2**, 312–318. Reprinted in Rosen [1967], pp. 221–228.

Dijkstra, E. W. [1963]. "An Algol 60 translator for the X1." In *Annual Review in Automatic Programming* **3**, Pergamon Press, New York, 329–345.

Dijkstra, E. W., and H. R. Mc Lean [1967]. "Register allocation for the C machine stack machine." *Proc. ACM Symposium on Principles of Programming Languages and Operating Systems*, 106–116.

Downey, P. J., and R. Sethi [1978]. "Assignment commands with array references." *J. ACM* **25**(4), 652–666.

Downey, P. J., R. Sethi, and R. L. Tarjan [1980]. "Variations on the common subexpression problem." *J. ACM* **27**(4), 758–771.

Earley, J. [1970]. "An efficient context-free parsing algorithm." *Comm. ACM* **13**(2), 94–102.

Earley, J. [1975a]. "Ambiguity and precedence in syntax description." *Acta Informatica* **4**(2), 183–192.

Earley, J. [1975b]. "High level iterators and a method for data structure abstraction." *J. Computer Languages* **1**(4), 321–342.

Elson, M., J. L. Rake [1970]. "An analysis of common-case PL/I programs." *ACM Trans. Software Engineering* **SE2.2**, 113–120.

Endlich, [1981]. "Another exhaustive matcher." In Locke [1984], pp. 100–138.

Ershov, A. P. [1958]. "On programming of arithmetic operations." *Comm. ACM* **1**(8). Also in *Annual Review in Automatic Programming* **1**, Pergamon Press, New York, 1960, p. 3.

Ershov, A. P. [1966]. "ALPHA — an automatic programming system of high efficiency." *J. ACM* **13**(1), 17–24.

Ershov, A. P. [1971]. *The Alpha Automatic Programming System*. Academic Press, New York.

Fischer, A. P., and C. H. A. Koster [1973]. *Methods of Algorithmic Language Implementation*. Lecture Notes in Computer Science 47, Springer-Verlag, Berlin.

Fisher, [1972]. "TOLDS, a declarative formal language definition system." STAN-CS-72-329, Stanford Univ.

Fraser, [1984]. "Generating a code-location compiler from an attribute grammar." *IEEE Software* **1**(3), 73–83.

Garrow, R., and D. Yellin [1985]. "A comparison of storage optimizations in automatically generated compilers." Unpublished, Columbia Univ.

Gelernter, S. [1976a]. "Make — a program for maintaining computer programs." *Software—Practice and Experience* **9**(4), 255–265.

Gelernter, S. [1976b]. "Implementation of a portable Fortran 77 compiler using modern tools." *ACM SIGPLAN Notices* **14**(8), 98–104.

Gentleman, M. [1972]. "Efficiency of equivalence algorithms." In Miller and Thatcher [1972], pp. 153–168.

Gick, A. C. [1978]. "The impossibility of content exchange diffusion." *IEEE Symposium on Security and Privacy*, 11–13 (November 78-4).

Gimpel, R. W. [1961]. "An algorithm for coding efficient arithmetic expressions." *Comm. ACM* **4**(8), 352–361.

Floyd, R. W. [1963]. "Syntactic analysis and operator precedence." *J. ACM* **10**(3), 316–333.

Floyd, R. W. [1964]. "Bounded-context syntactic analysis." *Comm. ACM* **7**(2), 62–67.

Aho, A. C. [1979]. "Automatic implementation of programs in very high level languages." *Sixth Annual ACM Symposium on Principles of Programming Languages*, 21–38.

Aho, A. C., and J. D. Ullman [1976]. "Indistinguishable variables in very high-level languages." *Third Annual ACM Symposium on Principles of Programming Languages*, 104–112.

Ashcroft, E. D., and J. Osterweil [1976]. "Data flow analysis in software reliability." *Computing Surveys* 8:3, 305–330.

Bagley, J. M. [1968]. "A syntax-improving program." *Commun. ACM* 11:1, 31–34.

Banner, C. W. [1977]. *Automatic Generation of Code Generators*. Ph. D. Thesis, Yale Univ., New Haven, Conn.

Beaver, C. W. [1979]. "A compact, machine-independent peephole optimizer." *Sixth Annual ACM Symposium on Principles of Programming Languages*, 1–6.

Beaver, C. W., and D. R. Hanson [1982]. "A machine-independent linker." *Software—Practice and Experience* 12, 351–366.

Bentley, M. L., J. Knotek, and L. Szpiratka [1984]. "Steiner trees in table with QUBE-oriented geometries." *ACM IEEE*, 539–544.

Bird, C. [1979]. "Bigraphs: a formal language, modeled upon that of arithmetic, for pure research." *Information* [1967], 1–65.

Blair-Johnson, R. A. [1969]. "The Multics PL/I compiler." *AFIPS Fall Joint Computer Conference*, 35, 187–208.

Brelaz, D. [1979]. "Register allocation on a single-chip." *Commun. ACM* 17:11, 638–642.

Brooks-Shirley, S. M. [1984]. "On the use of global optimization algorithms for the detection of semantic programming errors." SMU-34. New York Univ.

Brooks-Shirley, S. M., J. T. Schwartz, and M. Sharir [1983]. "Experience with the SETL optimizer." *TOPLAS* 5:1, 26–45.

Bumgarner, H. [1975]. "Some statistics on the usage of the C language." AT&T Bell Laboratories, Murray Hill, N. J.

Cattell, R. A., and M. J. Fischer [1984]. "An improved sequential algorithm." *Comm. of ACM* 7:5, 311–403.

Chandrasekar, M. [1980]. *Retargetable Code Generation using Attributed Grammars*. Ph. D. Thesis, Univ. of Wisconsin, Madison, Wis.

Christoph, M., and C. N. Fischer [1982]. "Description-driven code generation using attribute grammars." *Sixth ACM Symposium on Principles of Programming Languages*, 108–119.

Christopher, M., C. N. Fischer, and T. L. Henderson [1982]. "Retargetable compiler code generation." *Computing Surveys* 14:4, 573–592.

Clarkson, J. D., and J. J. Horning [1975]. "Language design for programming reliability." *IEEE Trans. Software Engineering* SE-1:2, 179–191.

Cocke, R., R. Gillett, J. Mocenigo, and R. Witek [1982]. "A study of formal semantics directed compiler generation." *ACM SIGPLAN Notices* 17:6, June, 172–184.

Cocke, R., and K. Kennedy [1981]. "Optimization of Ada." *ACM SIGPLAN Notices* 15:2, February, 30–42.

Cody, M. R., and D. S. Johnson [1979]. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco.

Cole, C. W. [1965]. "A proposed compilation of efficient object code." *Commun. ACM* 8:8, 483–488.

Cousot, C. M. [1972]. *Global Program Optimization*. Ph. D. Thesis, Dept. of Computer Science, Carnegie-Mellon Univ.

Cousot, R. R. [1983]. "A formal framework for the derivation of machine-specific optimizers." *TOPLAS* 5:3, 337–448.

Cousot, R., and R. Williams [1978]. "Counter-one pass features in one-pass compilation: a formalization using attribute grammars." *Information Processing Letters* 7:6, 270–284.

Cousot, R. S. [1981]. *Automatic Interprocedural Algorithms for Code Generation*. *Code Generation*. Ph. D. Thesis, Univ. of California, Berkeley.

Cousot, R. S., and C. Cousot [1978]. "A new method for compiler code generation." *Eighth ACM Symposium on Principles of Programming Languages*, 231–240.

Cousot, R. M. [1981]. "Register allocation complexity." *IEEE Software Engineering* 48, 305–317. Reprinted in Rosen [1967], pp. 284–305.

Cousot, N. L. [1968]. "Table driven code generation." *Computer* 1358, 27–35.

Cousot, S. L. [1984]. "Code generation and optimization." in Cocke [1981], pp. 251–268.

Cousot, N. L., C. B. Cousot, and W. N. Hall [1976]. "Practical LR error recovery." *ACM SIGPLAN Notices* 14:8, 168–175.

Cousot, S. L., M. A. Harrison, and W. L. Ruzzo [1981]. "An improved context-free recognizer." *TOPLAS* 2:3, 415–462.

Cousot, N. L., and S. D. Brelaz [1975]. "Practical syntactic error recovery." *Comm. ACM* 18:11, 635–650.

Cousot, S. L., and K. Williams [1971]. "An Easily Implemented Algorithm for global data flow analysis." *J. ACM* 23:1, 172–202.

Cousot, A. A., J. Hell, and H. Lamport [1987]. *Compilation of Algol 60*. Springer-Verlag, New York.

Hanson, D. R. [1981]. "Is block structure necessary?" *Software—Practice and Experience* 11, 853–866.

Harrison, S. C. [1977]. "Implementation of the substring test by hashing." *Comm. ACM* 14:12, 777–779.

Harrison, W. [1977]. "Compiler analysis of the value ranges for variables." *IEEE Trans. Software Engineering* 3:3.

Hecht, M. S. [1977]. *Flow Analysis of Computer Programs*. North-Holland, New York.

Hecht, M. S., and J. B. Shaffer [1975]. "Ideas on the design of a Quick Optimizer for SEMICT programs and management analysis." Dept. of Computer Science, Univ. of Maryland, College Park, Md.

Hecht, M. S., and J. D. Ullman [1972]. "Flow graph reducibility." *SIAM J. Computing* 1, 188–202.

Hecht, M. S., and J. D. Ullman [1974]. "Characterizations of reducible flow graphs." *J. ACM* 21, 367–375.

Hecht, M. S., and J. D. Ullman [1975]. "A simple algorithm for global data flow analysis programs." *SIAM J. Computing* 4:5, 519–532.

Hennessy, J. [1981]. *Program Optimization and Exception Handling*. Harvard Univ. Press, Cambridge, Mass.

Hennessy, J. [1981]. "Program optimization and exception handling." *Eighth Annual ACM Symposium on Principles of Programming Languages*, 220–231.

Hennessy, J. [1982]. "Symbolic debugging of optimized code." *TOPLAS* 4:3, 323–344.

Henry, R. R. [1984]. *Graham-Glanville Code Generators*. Ph. D. Thesis, Univ. of California, Berkeley.

Hext, J. B. [1967]. "Compile time type-matching." *Computer J.* 9, 365–369.

HINDLEY, R. [1969]. "The principal type-scheme of an object in combinatory logic." *Trans. AMS* 146, 29–60.

HOPGOOD, F. R. A. [1969a]. "Quicksort." *Computer J.* 5:1, 44–45.

HOPGOOD, F. R. A. [1969b]. "Report on the Flavia Algol Translator." *Computer J.* 5:2, 127–129.

HOPCROFT, J. E. AND M. J. O'DONNELL [1969]. "Performance modifications of pushdown automata." *J. ACM* 16:2, 274–291.

HOPCROFT, J. E. AND R. M. KARP [1971]. "A linear algorithm for testing equivalence of finite automata." TR-71-114, Dept. of Computer Science, Cornell Univ. See also Hopcroft and Ullman [1974], pp. 143–145.

HOPCROFT, J. E. AND J. D. ULLMAN [1969]. *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, Mass.

HOPCROFT, J. E. AND J. D. ULLMAN [1973]. "Set merging algorithms." *SIAM J. Computing* 2:3, 294–303.

HOPCROFT, J. E. AND J. D. ULLMAN [1979]. *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, Mass.

HORNING, J. J. [1971]. "Who should construct the compiler, and how much should the compiler do?" in Rosen and Eckel [1974].

HOWARD, J. P., R. M. KARP, R. E. MILLER, AND S. WINOGRAD [1966]. "Index register allocation." *J. ACM* 13:1, 43–96.

HUET, G. AND G. KAHN [1975]. *Proving and Improving Programs*, Proceedings IRIA Arc. et Senans, France.

HUET, G. AND J.-J. LÉVY [1979]. "Call by need computations in non-ambiguous linear term rewriting systems." Rapport de Recherche 359, INRIA, Le Chesnay, Recquencourt.

HUFFMAN, D. A. [1954]. "The synthesis of sequential machines." *J. Assembly Ass.* 257, 3–4, 161–190; 275–303.

HUNT, J. W. AND M. D. MCILROY [1976]. "An algorithm for differential file comparison." Computing Science Technical Report 41, AT&T Bell Laboratories, Murray Hill, N. J.

HUNT, J. W. AND T. G. SZYMANSKI [1977]. "A fast algorithm for computing longest common subsequences." *Comm. ACM* 20:5, 350–353.

HUSKEY, H. D., M. H. HALSTEAD, AND R. MCARTHUR [1960]. "Neliac — a dialect of Algol." *Comm. ACM* 3:8, 463–468.

IRONS, E. T. AND S. F. KIRSLIS [1970]. "A technique for generating a mixed optional Floyd-Evans production precedence parser." *Comm. ACM* 13:8, 501–508.

INGALLS, D. H. H. [1978]. "The Smalltalk-76 programming system: design and implementation." *Fifth Annual ACM Symposium on Principles of Programming Languages*, 9–16.

INGERMAN, P. Z. [1963]. "Panini-Backus form suggested." *Comm. ACM* 10:3, 137.

IRONS, E. T. [1961]. "A syntax directed compiler for Algol 60." *Comm. ACM* 4:1, 51–55.

IRONS, E. T. [1963]. "An error-correcting parse algorithm." *Comm. ACM* 6:11, 669–673.

IVERSON, K. [1962]. *A Programming Language*. Wiley, New York.

JAMES, J. M. [1981]. "A comment on 'Operator identification in ALL' by Gansinger and Ripken." *ACM SIGPLAN Notices* 15:9 (September), 39–41.

JARVIS, J. F. [1976]. "Feature recognition in line drawings using regular expressions." *Proc. 3rd Int. Joint Conf. on Pattern Recognition*, 189–192.

JAZAYERI, M., W. F. OGDEN, AND W. C. ROUNDS [1975]. "The intrinsic exponential complexity of the circularity problem for attribute grammars." *Comm. ACM* 18:12, 697–706.

JAZAYERI, M. AND D. POZEFSKY [1981]. "Space efficient storage management in attribute grammar evaluators." *TOPLAS* 3:4, 388–404.

JAZAYERI, M. AND K. G. WALTER [1975]. "Alternating semantic evaluator." *Proc. ACM Annual Conference*, 230–234.

JENSEN, K. AND N. WIRTH [1975]. *Pascal User Manual and Report*. Springer-Verlag, New York.

JOHNSON, S. L. [1975]. "Yacc — yet another compiler-compiler." Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, N. J.

JOHNSON, S. L. [1978]. "A portable compiler: theory and practice." *Fifth Annual ACM Symposium on Principles of Programming Languages*, 97–104.

JOHNSON, S. C. [1979]. "A tour through the portable C compiler." AT&T Bell Laboratories, Murray Hill, N. J.

JOHNSTON, S. C. [1981]. "Code generation for silicon." *Comp. Science and Statistics: Proc. 13th Symposium on the Interface*, 14–19.

JOHNSON, S. C. AND M. E. LESK [1978]. "Language development tools." *Bell System Technical J.* 57:6, 2155–2175.

JOHNSON, S. C. AND D. M. RITCHIE [1981]. "The C language calling sequence." Computing Science Technical Report 102, AT&T Bell Laboratories, Murray Hill, N. J.

JOHNSON, W. L., J. H. PORTER, S. I. ACKLEY, AND D. T. ROSS [1968]. "Automatic generation of efficient lexical processors using finite state techniques." *Comm. ACM* 11:12, 805–813.

JOHNSSON, R. K. [1975]. *An Approach to Global Register Allocation*. Ph. D. Thesis, Carnegie-Mellon Univ., Pittsburgh, Pa.

JONES, A. K. [1976]. "An eclectic scheme for partial evaluation of and projections over LISP-like Cartesian." *2nd Texas Conf. on Computer Systems*, 363–364.

JONES, N. D. [1980]. *Semantics-Directed Compiler Generation*, Lecture Notes in Computer Science 94, Springer-Verlag, Berlin.

JONES, N. D. AND C. M. MADSEN [1980]. "Attribute-influenced LR parsing." in Jones [1980], pp. 393–407.

JONES, N. D. AND S. S. MUCHNICK [1976]. "Binding time optimization in programming languages." *Third ACM Symposium on Principles of Programming Languages*, 77–94.

JOSEPHS, M. [1984]. "Strongly noncircular attribute grammars and their recursive evaluation." *ACM SIGPLAN Notices* 19:6, 91–99.

KAHN, G., D. B. MACQUEEN, AND G. PLOTKIN [1984]. *Semantics of Data Types*, Lecture Notes in Computer Science 173, Springer-Verlag, Berlin.

KAM, J. B. AND J. D. ULLMAN [1976]. "Global data flow analysis and iterative algorithms." *J. ACM* 23:1, 158–171.

KAM, J. B. AND J. D. ULLMAN [1977]. "Monotone data flow analysis frameworks." *Acta Informatica* 7:3, 305–318.

KAPLAN, M. AND J. D. ULLMAN [1980]. "A general scheme for the automatic inference of variable types." *J. ACM* 27:1, 128–145.

KASAMI, T. [1965]. "An efficient recognition and syntax analysis algorithm for context-free languages." AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Mass.

KASAMI, T., W. W. PETERSON, AND N. TOKURA [1973]. "On the capabilities of while, repeat, and exit statements." *Comm. ACM* 16:8, 503–512.

KASTENS, U. [1980]. "Ordered attribute grammars." *Acta Informatica* 13:3, 229–256.

KASTENS, U., B. HUTT, AND E. ZIMMERMANN [1982]. *GAG: A Practical Compiler Generator*. Lecture Notes in Computer Science 141, Springer-Verlag, Berlin.

KNUTH, V. A., [1971]. "Some properties of tally trees," *Information Processing Letters* 2:4, 113–115.

KNUTH, D. E. [1964]. "Translation of Gauss's elimination method into procedures," *SIGNUM* 6:3, 5–9.

KOSARAJU, R., [1971]. "A global flow analysis algorithm," *International J. Computer and Information Sciences* 4:3, 214–15.

KENNEDY, K. [1972]. "Index register allocation in straight line code and simple loops," in Rustin [1972], pp. 51–64.

KENNEDY, K. [1976]. "A comparison of two algorithms for global flow analysis," *SIAM J. Computing* 5:1, 158–180.

KENNEDY, K. [1981]. "A survey of data flow analysis techniques," in Muchnick and Jones [1981], pp. 5–54.

KENNEDY, K., AND J. RAMANATHAN [1979]. "A deterministic attribute grammar evaluator based on dynamic sequencing," *TOPLAS* 1:1, 142–160.

KENNEDY, K., AND S. K. WARREN [1976]. "Automatic generation of efficient evaluators for attribute grammars," *Conf. Rec. 3rd Symposium on Principles of Programming Languages*, 32–49.

KERNIGHAN, B. W., [1975]. "Ratfor — a preprocessor for a rational Fortran," *Software — Practice and Experience* 5:4, 395–406.

KERNIGHAN, B. W., [1982]. "PIC — a language for typesetting graphics," *Software — Practice and Experience* 12:1, 1–21.

KERNIGHAN, B. W., AND L. L. COWAN [1976]. "A system for typesetting mathematics," *Comm. ACM* 18:3, 151–157.

KERNIGHAN, B. W., AND R. PIKE [1984]. *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, N.J.

KERNIGHAN, B. W., AND D. M. RITCHIE [1978]. *The C Programming Language*, Prentice-Hall, Englewood Cliffs, N.J.

KILDALL, G. [1973]. "A unified approach to global program optimization," *ACM Symposium on Principles of Programming Languages*, 194–206.

KLEENE, S. C. [1956]. "Representation of events in nerve nets," in Shannon and McCarthy [1956], pp. 3–40.

KNUTH, D. E. [1962]. "A history of writing compilers," *Computers and Automation*, December 1962, 8–18. Reprinted in Pollack [1972], pp. 38–56.

KNUTH, D. E. [1964]. "Backus Normal Form vs. Backus Naur Form," *Comm. ACM* 7:12, 735–736.

KNUTH, D. E. [1965]. "On the translation of languages from left to right," *Information and Control* 8:6, 607–639.

KNUTH, D. E. [1968]. "Semantics of context-free languages," *Mathematical Systems Theory* 2:2, 127–145. Errata in 5:1 [1971] 95–96.

KNUTH, D. E. [1971a]. "Top-down syntax analysis," *Acta Informatica* 1:2, 79–110.

KNUTH, D. E. [1971b]. "An empirical study of FORTRAN programs," *Software — Practice and Experience* 1:2, 105–133.

KNUTH, D. E. [1973a]. *The Art of Computer Programming*, Vol. 1, 2nd Ed. *Fundamental Algorithms*, Addison-Wesley, Reading, Mass.

KNUTH, D. E. [1973b]. *The Art of Computer Programming*, Vol. 3, *Sorting and Searching*, Addison-Wesley, Reading, Mass.

KNUTH, D. E. [1977]. "A generalization of Dijkstra's algorithm," *Information Processing Letters* 6:1, 1–5.

KNUTH, D. E. [1984]. *The TEXbook*, Addison-Wesley, Reading, Mass.

KNUTH, D. E. [1985]. "Dynamic Huffman coding," *J. Algorithms* 6:2, 163–180.

KNUTH, D. E., J. H. MORRIS, AND V. R. PRATT [1977]. "Fast pattern matching in strings," *SIAM J. Computing* 6:2, 323–350.

KNUTH, D. E., AND L. TRABB PARDO [1977]. "Early development of programming languages," *Encyclopedia of Computer Science and Technology* 7, Marcel Dekker, New York, 419–493.

KOSTELAK, A. J. [1992]. "A practical introduction to LR(k) parsers," *Comm. ACM* 35:11, 63–67.

KOSTAS, S. R. [1974]. "Analysis of unmatched arguments," *J. Computer and Software Sciences* 9:3, 353–358.

KOSTOULAS, N. AND K. T. RAYNER [1983]. "Modelling of space-efficient one-pass translation using attribute grammars," *Software — Practice and Experience* 13, 1119–1129.

KOSTER, C. H. A. [1971]. "Affix grammars," in Peck [1971], pp. 95–109.

KOU, L. [1977]. On live-dead analysis for global data flow problems," *JACM* 24:3, 473–483.

KRASNESKY, B. AND S. JAMSHIDI [1981]. "Methods for recognizing LALR(1) look-backs," *TOPLAS* 3:1, 60–82.

KROS, H. [1975]. *Two Templates and Subset Languages for Environmental Grammars*, Ph.D. Thesis, U. of California, Santa Cruz.

LALONDE, W. R. [1971]. "An efficient LALR parser generator," Tech. Rep. 2, Computer Systems Research Group, Univ. of Toronto.

LALONDE, W. R. [1976]. "On directly constructing LR(k) parsers without using productions," *Proc. 8th Symposium on Principles of Programming Languages*, 127–133.

LAM, J. W. R., E. S. LEE, AND J. J. HORNING [1971]. "XPL/1, A Rapid Response Compiler," *Proc. IFIP Congress 71*, TA-3, North-Holland, Amsterdam, 197–201.

LAMB, D. A. [1981]. "Construction of a peephole optimizer," *Software — Practice and Experience* 11, 638–647.

LAMPSON, B. W., [1982]. "Fast procedure calls," *SIGPLAN Notices* 17:8 (Aug.), 66–76.

LANDIN, P. J. [1964]. "The mechanical evaluation of expressions," *Computer J.* 6:4, 308–320.

LEAVENWORTH, B. AND M. C. PEREIRA [1968]. "Set complexity considerations in applications of their implementation and portability," *Software — Practice and Experience* 8, 149–159.

LEHMANN, H. P. [1977]. "Let must languages — a study of special issues in programming languages," *Computing Surveys* 9:3, 195–210.

LEVIEN, R. E. [1970]. *Linear Pattern Matching for Syntax Directed Compiler Systems*, Ph.D. Thesis, University of Wisconsin, Madison.

LEWIS, P. M. II, D. J. ROSENKRANTZ, AND R. E. STEARNS [1974]. "Attributed translations for finitely ambiguous grammars in a flowgraph," *TOPLAS* 1, 131–141.

LESK, M. E. [1975]. "Lex — a lexical analyzer generator," Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J.

LEVERETT, H. W. [1981]. "Topics in code generation and register allocation," CMU-CS-82-130, Computer Science Dept., Carnegie Mellon Univ., Pittsburgh, Pennsylvania.

LEVERETT, B. W., R. G. G. CATTELL, S. O. HOBBS, J. M. NEWCOMER, A. H. REINER, B. R. SCHATZ, AND W. A. WULF [1980]. "An overview of the production-quality compiler-compiler project," *Computer* 13:8, 38–49.

LEVERETT, H. W. AND T. G. SZYMANSKI [1980]. "Chaining span-dependent jump instructions," *TOPLAS* 2:3, 274–289.

LEVY, J. P. [1975]. "Automatic correction of syntax errors in programming languages," *Acta Informatica* 4, 271–292.

[1] ADAMS, D. A. [1967] "A computation procedure for inventory models with sequence dependent setup costs," *Management Science* 14, 43–60.

[2] AHO, A. V., J. E. HOPCROFT, AND J. D. ULLMAN [1974] *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass.

[3] AHO, A. V., S. C. JOHNSON, AND J. D. ULLMAN [1975] "Deterministic parsing of ambiguous grammars," *Communications of the ACM* 18, 441–452.

[4] AHO, A. V., R. SETHI, AND J. D. ULLMAN [1986] *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Mass.

[5] AHO, A. V., AND J. D. ULLMAN [1972] *The Theory of Parsing, Translation, and Compiling*, Vol. I: *Parsing*, Prentice-Hall, Englewood Cliffs, N. J.

[6] AHO, A. V., AND J. D. ULLMAN [1973] *The Theory of Parsing, Translation, and Compiling*, Vol. II: *Compiling*, Prentice-Hall, Englewood Cliffs, N. J.

[7] AHO, A. V., AND J. D. ULLMAN [1977] *Principles of Compiler Design*, Addison-Wesley, Reading, Mass.

[8] AHO, A. V., AND S. C. JOHNSON [1974] "LR parsing," *Computing Surveys* 6, 99–124.

[9] AHO, A. V., S. C. JOHNSON, AND J. D. ULLMAN [1975] "Code generation for expressions with common subexpressions," *Journal of the ACM* 24, 146–160.

[10] AHO, A. V., S. C. JOHNSON, AND J. D. ULLMAN [1977] "Code generation for machines with multiregister operations," *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, 21–28.

[11] AHO, A. V., R. SETHI, AND J. D. ULLMAN [1981] "A fast algorithm for finding nearest common ancestors," *SIAM Journal on Computing* 12, 1–12.

[12] AHO, A. V., AND T. G. SZYMANSKI [1977] "Recognizing noncontext-free languages," *Journal of the ACM* 24, 96–100.

[13] AHO, A. V., AND J. D. ULLMAN [1972] "Optimization of straight line programs," *SIAM Journal on Computing* 1, 1–19.

[14] AHO, A. V., AND J. D. ULLMAN [1973] "A technique for speeding up LR(k) parsers," *Conference Record of the Fifth Annual ACM Symposium on Theory of Computing*, 251–263.

[15] AHO, A. V., AND J. D. ULLMAN [1973] "Node listings for reducible flow graphs," *Conference Record of the Seventh Annual ACM Symposium on Theory of Computing*, 177–185.

[16] AHO, A. V., AND J. D. ULLMAN [1977] *Principles of Compiler Design*, Addison-Wesley, Reading, Mass.

[17] AHO, A. V., AND J. D. ULLMAN [1979] "Universality of data retrieval languages," *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, 110–120.

[18] AHO, A. V., AND J. D. ULLMAN [1979] "A characterization of regular languages that notate transductions," *SIAM Journal on Computing* 8, 708–725.

[19] AHO, A. V., AND J. D. ULLMAN [1979] "Optimization of straight line programs," *SIAM Journal on Computing* 8, 1–27.

[20] AHO, A. V., AND J. D. ULLMAN [1979] "Probability of a tree being a subtree," *Information Processing Letters* 8, 12–13.

[21] AHO, A. V., AND J. D. ULLMAN [1979] "The theory of languages," *Mathematical Systems Theory* 12, 97–125.

[22] AHO, A. V., AND J. D. ULLMAN [1981] "Node listings for reducible flow graphs," *Journal of Computer and System Sciences* 21, 21–38.

[23] AHO, A. V., AND M. ULLMAN [1983] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[24] AHO, A. V., AND M. ULLMAN [1985] "Universality of data retrieval languages," *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, 110–117.

[25] AHO, A. V., AND M. ULLMAN [1986] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[26] AHO, A. V., AND M. ULLMAN [1987] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[27] AHO, A. V., AND M. ULLMAN [1988] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[28] AHO, A. V., AND M. ULLMAN [1989] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[29] AHO, A. V., AND M. ULLMAN [1990] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[30] AHO, A. V., AND M. ULLMAN [1991] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[31] AHO, A. V., AND M. ULLMAN [1992] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[32] AHO, A. V., AND M. ULLMAN [1993] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[33] AHO, A. V., AND M. ULLMAN [1994] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[34] AHO, A. V., AND M. ULLMAN [1995] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[35] AHO, A. V., AND M. ULLMAN [1996] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[36] AHO, A. V., AND M. ULLMAN [1997] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[37] AHO, A. V., AND M. ULLMAN [1998] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[38] AHO, A. V., AND M. ULLMAN [1999] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[39] AHO, A. V., AND M. ULLMAN [2000] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[40] AHO, A. V., AND M. ULLMAN [2001] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[41] AHO, A. V., AND M. ULLMAN [2002] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[42] AHO, A. V., AND M. ULLMAN [2003] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[43] AHO, A. V., AND M. ULLMAN [2004] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[44] AHO, A. V., AND M. ULLMAN [2005] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[45] AHO, A. V., AND M. ULLMAN [2006] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[46] AHO, A. V., AND M. ULLMAN [2007] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[47] AHO, A. V., AND M. ULLMAN [2008] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[48] AHO, A. V., AND M. ULLMAN [2009] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[49] AHO, A. V., AND M. ULLMAN [2010] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[50] AHO, A. V., AND M. ULLMAN [2011] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[51] AHO, A. V., AND M. ULLMAN [2012] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[52] AHO, A. V., AND M. ULLMAN [2013] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[53] AHO, A. V., AND M. ULLMAN [2014] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[54] AHO, A. V., AND M. ULLMAN [2015] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[55] AHO, A. V., AND M. ULLMAN [2016] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[56] AHO, A. V., AND M. ULLMAN [2017] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[57] AHO, A. V., AND M. ULLMAN [2018] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[58] AHO, A. V., AND M. ULLMAN [2019] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[59] AHO, A. V., AND M. ULLMAN [2020] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[60] AHO, A. V., AND M. ULLMAN [2021] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[61] AHO, A. V., AND M. ULLMAN [2022] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[62] AHO, A. V., AND M. ULLMAN [2023] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[63] AHO, A. V., AND M. ULLMAN [2024] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[64] AHO, A. V., AND M. ULLMAN [2025] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[65] AHO, A. V., AND M. ULLMAN [2026] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[66] AHO, A. V., AND M. ULLMAN [2027] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[67] AHO, A. V., AND M. ULLMAN [2028] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[68] AHO, A. V., AND M. ULLMAN [2029] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[69] AHO, A. V., AND M. ULLMAN [2030] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[70] AHO, A. V., AND M. ULLMAN [2031] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[71] AHO, A. V., AND M. ULLMAN [2032] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[72] AHO, A. V., AND M. ULLMAN [2033] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[73] AHO, A. V., AND M. ULLMAN [2034] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[74] AHO, A. V., AND M. ULLMAN [2035] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[75] AHO, A. V., AND M. ULLMAN [2036] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[76] AHO, A. V., AND M. ULLMAN [2037] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[77] AHO, A. V., AND M. ULLMAN [2038] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[78] AHO, A. V., AND M. ULLMAN [2039] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[79] AHO, A. V., AND M. ULLMAN [2040] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[80] AHO, A. V., AND M. ULLMAN [2041] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[81] AHO, A. V., AND M. ULLMAN [2042] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[82] AHO, A. V., AND M. ULLMAN [2043] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[83] AHO, A. V., AND M. ULLMAN [2044] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[84] AHO, A. V., AND M. ULLMAN [2045] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[85] AHO, A. V., AND M. ULLMAN [2046] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[86] AHO, A. V., AND M. ULLMAN [2047] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[87] AHO, A. V., AND M. ULLMAN [2048] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[88] AHO, A. V., AND M. ULLMAN [2049] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[89] AHO, A. V., AND M. ULLMAN [2050] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[90] AHO, A. V., AND M. ULLMAN [2051] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[91] AHO, A. V., AND M. ULLMAN [2052] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[92] AHO, A. V., AND M. ULLMAN [2053] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[93] AHO, A. V., AND M. ULLMAN [2054] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[94] AHO, A. V., AND M. ULLMAN [2055] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[95] AHO, A. V., AND M. ULLMAN [2056] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[96] AHO, A. V., AND M. ULLMAN [2057] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[97] AHO, A. V., AND M. ULLMAN [2058] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[98] AHO, A. V., AND M. ULLMAN [2059] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[99] AHO, A. V., AND M. ULLMAN [2060] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[100] AHO, A. V., AND M. ULLMAN [2061] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[101] AHO, A. V., AND M. ULLMAN [2062] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[102] AHO, A. V., AND M. ULLMAN [2063] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[103] AHO, A. V., AND M. ULLMAN [2064] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[104] AHO, A. V., AND M. ULLMAN [2065] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[105] AHO, A. V., AND M. ULLMAN [2066] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[106] AHO, A. V., AND M. ULLMAN [2067] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[107] AHO, A. V., AND M. ULLMAN [2068] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[108] AHO, A. V., AND M. ULLMAN [2069] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[109] AHO, A. V., AND M. ULLMAN [2070] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[110] AHO, A. V., AND M. ULLMAN [2071] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[111] AHO, A. V., AND M. ULLMAN [2072] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[112] AHO, A. V., AND M. ULLMAN [2073] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[113] AHO, A. V., AND M. ULLMAN [2074] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[114] AHO, A. V., AND M. ULLMAN [2075] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[115] AHO, A. V., AND M. ULLMAN [2076] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[116] AHO, A. V., AND M. ULLMAN [2077] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[117] AHO, A. V., AND M. ULLMAN [2078] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[118] AHO, A. V., AND M. ULLMAN [2079] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[119] AHO, A. V., AND M. ULLMAN [2080] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[120] AHO, A. V., AND M. ULLMAN [2081] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[121] AHO, A. V., AND M. ULLMAN [2082] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[122] AHO, A. V., AND M. ULLMAN [2083] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[123] AHO, A. V., AND M. ULLMAN [2084] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[124] AHO, A. V., AND M. ULLMAN [2085] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[125] AHO, A. V., AND M. ULLMAN [2086] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[126] AHO, A. V., AND M. ULLMAN [2087] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[127] AHO, A. V., AND M. ULLMAN [2088] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[128] AHO, A. V., AND M. ULLMAN [2089] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[129] AHO, A. V., AND M. ULLMAN [2090] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[130] AHO, A. V., AND M. ULLMAN [2091] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[131] AHO, A. V., AND M. ULLMAN [2092] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[132] AHO, A. V., AND M. ULLMAN [2093] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[133] AHO, A. V., AND M. ULLMAN [2094] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[134] AHO, A. V., AND M. ULLMAN [2095] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[135] AHO, A. V., AND M. ULLMAN [2096] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[136] AHO, A. V., AND M. ULLMAN [2097] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[137] AHO, A. V., AND M. ULLMAN [2098] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[138] AHO, A. V., AND M. ULLMAN [2099] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

[139] AHO, A. V., AND M. ULLMAN [2100] "Unification of polymorphic functions," *Theoretical Computer Science* 25, 267–310.

Paige, R. and J. T. Schwartz [1977]. "Expression continuity and the formal differentiation of algorithms," *Fourth ACM Symposium on Principles of Programming Languages*, 58–71.

Palm, R. C., Jr. [1975]. "A portable optimizer for the language C," M. Sc. Thesis, MIT, Cambridge, Mass.

Park, J. C. H., K. M. Choe, and C. H. Chang [1985]. "A new analysis of LALR formalisms," *TOPLAS* 7:1, 159–175.

Paterson, M. S. and M. Wegman [1978]. "Linear unification," *J. Computer and System Sciences* 16:2, 158–167.

Peck, J. E. L. [1971]. *Algol 68 Implementation*, North-Holland, Amsterdam.

Pennello, T. and F. DeRemer [1978]. "A forward move algorithm for LR error recovery," *Fifth Annual ACM Symposium on Principles of Programming Languages*, 241–254.

Pennello, T., F. DeRemer, and R. Meyers [1980]. "A simplified operator identification scheme for Ada," *ACM SIGPLAN Notices* 15:7 (July–August) 82–87.

Persch, G., G. Winterstein, M. Dausmann, and S. Drossopoulou [1980]. "Overloading in preliminary Ada," *ACM SIGPLAN Notices* 15:11 (November) 47–56.

Peterson, W. W. [1957]. "Addressing for random access storage," *IBM J. Research and Development* 1:2, 130–146.

Pollack, B. W. [1972]. *Compiler Techniques*, Auerbach Publishers, Princeton, N.J.

Pollock, L. L. and M. L. Soffa [1985]. "Incremental compilation of locally optimized code," *Twelfth Annual ACM Symposium on Principles of Programming Languages*, 152–164.

Powell, M. L. [1984]. "A portable optimizing compiler for Modula-2," *ACM SIGPLAN Notices* 19:6, 310–318.

Pratt, T. W. [1984]. *Programming Languages: Design and Implementation*, 2nd Ed., Prentice-Hall, Englewood Cliffs, N.J.

Pratt, V. R. [1973]. "Top down operator precedence," *ACM Symposium on Principles of Programming Languages*, 41–51.

Price, C. E. [1971]. "Table lookup techniques," *Computing Surveys* 3:2, 49–65.

Prosser, R. T. [1959]. "Applications of boolean matrices to the analysis of flow diagrams," *AFIPS Eastern Joint Computer Conf.*, Spartan Books, Baltimore Md., 133–138.

Purdom, P. and C. A. Brown [1980]. "Semantic routines and LR(k) parsers," *Acta Informatica* 14:4, 299–315.

Purdom, P. W. and E. F. Moore [1972]. "Immediate predominators in a directed graph," *Comm. ACM* 15:8, 777–778.

Rabin, M. O. and D. Scott [1959]. "Finite automata and their decision problems," *IBM J. Research and Development* 3:2, 114–125.

Radin, G. and H. P. Rogoway [1965]. "NPL: Highlights of a new programming language," *Comm. ACM* 8:1, 9–17.

Räihä, K.-J. [1981]. *A Space Management Technique for Multi-Pass Attribute Evaluators*, Ph.D. Thesis, Report A-1981-4, Dept. of Computer Science, University of Helsinki.

Räihä, K.-J. and M. Saarinen [1982]. "Testing attribute grammars for circularity," *Acta Informatica* 17, 185–192.

Räihä, K.-J., M. Saarinen, M. Sarjakoski, S. Sippu, E. Soisalon-Soininen, and M. Tienari [1983]. "Revised report on the compiler writing system HLP78," Report A-1983-1, Dept. of Computer Science, University of Helsinki.

Randell, B. and L. J. Russell [1964]. *Algol 60 Implementation*, Academic Press, New York.

Redziejowski, R. R. [1969]. "On arithmetic expressions and trees," *Comm. ACM* 12:2, 81–84.

Reif, J. H. and H. R. Lewis [1977]. "Symbolic evaluation and the global value graph," *Fourth ACM Symposium on Principles of Programming Languages*, 104–118.

Reiss, S. P. [1983]. "Generation of compiler symbol processing mechanisms from specifications," *TOPLAS* 5:2, 127–163.

Reps, T. W. [1984]. *Generating Language-Based Environments*, MIT Press, Cambridge, Mass.

Reynolds, J. C. [1985]. "Three approaches to type structure," *Mathematical Foundations of Software Development*, Lecture Notes in Computer Science **185**, Springer-Verlag, Berlin, 97–138.

Richards, M. [1971]. "The portability of the BCPL compiler," *Software—Practice and Experience* 1:2, 135–146.

Richards, M. [1977]. "The implementation of the BCPL compiler," in P. J. Brown (ed.), *Software Portability: An Advanced Course*, Cambridge University Press.

Ripken, K. [1977]. "Formale Beschreibung von Maschinen, Implementierungen und optimierender Maschinencodeerzeugung aus attributierten Programmgraphen," TUM-INFO-7731, Institut für Informatik, Universität München, Munich.

Ripley, G. D. and F. C. Druseikis [1978]. "A statistical analysis of syntax errors," *Computer Languages* 3, 227–240.

Ritchie, D. M. [1979]. "A tour through the UNIX C compiler," AT&T Bell Laboratories, Murray Hill, N.J.

Ritchie, D. M. and K. Thompson [1974]. "The UNIX time-sharing system," *Comm. ACM* **17**:7, 365–375.

Robertson, E. L. [1979]. "Code generation and storage allocation for machines with span-dependent instructions," *TOPLAS* 1:1, 71–83.

Robinson, J. A. [1965]. "A machine-oriented logic based on the resolution principle," *J. ACM* **12**:1, 23–41.

Rohl, J. S. [1975]. *An Introduction to Compiler Writing*, American Elsevier, New York.

Rohrich, J. [1980]. "Methods for the automatic construction of error correcting parsers," *Acta Informatica* 13:2, 115–139.

Rosen, B. K. [1977]. "High-level data flow analysis," *Comm. ACM* **20**, 712–724.

Rosen, B. K. [1980]. "Monoids for rapid data flow analysis," *SIAM J. Computing* **9**:1, 159–196.

Rosen, S. [1967]. *Programming Systems and Languages*, McGraw-Hill, New York.

Rosenkrantz, D. J. and R. E. Stearns [1970]. "Properties of deterministic top-down grammars," *Information and Control* 17:3, 226–256.

Rosler, L. [1984]. "The evolution of C — past and future," *AT&T Bell Labs Technical Journal* **63**:8, 1685–1699.

Rustin, R. [1972]. *Design and Optimization of Compilers*, Prentice-Hall, Englewood Cliffs, N.J.

Ryder, B. G. [1979]. "Constructing the call graph of a program," *IEEE Trans. Software Engineering* **SE-5**:3, 216–226.

Ryder, B. G. [1983]. "Incremental data flow analysis," *Tenth ACM Symposium on Principles of Programming Languages*, 167–176.

Saarinen, M. [1978]. "On constructing efficient evaluators for attribute grammars," *Automata, Languages and Programming, Fifth Colloquium*, Lecture Notes in Computer Science **62**, Springer-Verlag, Berlin, 382–397.

SAMELSON, K. AND F. L. BAUER [1960]. "Sequential formula translation," *Comm. ACM* **3**:2, 76-83.

SANKOFF, D. AND J. B. KRUSKAL (EDS.) [1983]. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, Mass.

SCARBOROUGH, R. G. AND H. G. KOLSKY [1980]. "Improved optimization of FORTRAN object programs," *IBM J. Research and Development* **24**:6, 660-676.

SCHAEFER, M. [1973]. *A Mathematical Theory of Global Program Optimization*, Prentice-Hall, Englewood Cliffs, N. J.

SCHONBERG, E., J. T. SCHWARTZ, AND M. SHARIR [1981]. "An automatic technique for selection of data representations in SETL programs," *TOPLAS* **3**:2, 126-143.

SCHORRE, D. V. [1964]. "Meta-II: a syntax-oriented compiler writing language," *Proc. 19th ACM National Conf.*, D1.3-1–D1.3-11.

SCHWARTZ, J. T. [1973]. *On Programming: An Interim Report on the SETL Project*, Courant Inst., New York.

SCHWARTZ, J. T. [1975a]. "Automatic data structure choice in a language of very high level," *Comm. ACM* **18**:12, 722-728.

SCHWARTZ, J. T. [1975b]. "Optimization of very high level languages," *Computer Languages*. Part I: "Value transmission and its corollaries," **1**:2, 161-194; part II: "Deducing relationships of inclusion and membership," **1**:3, 197-218.

SEDGEWICK, R. [1978]. "Implementing Quicksort programs," *Comm. ACM* **21**, 847-857.

SETHI, R. [1975]. "Complete register allocation problems," *SIAM J. Computing* **4**:3, 226-248.

SETHI, R. AND J. D. ULLMAN [1970]. "The generation of optimal code for arithmetic expressions," *J. ACM* **17**:4, 715-728.

SHANNON, C. AND J. MCCARTHY [1956]. *Automata Studies*, Princeton University Press.

SHERIDAN, P. B. [1959]. "The arithmetic translator-compiler of the IBM Fortran automatic coding system," *Comm. ACM* **2**:2, 9-21.

SHIMASAKI, M., S. FUKUYA, K. IKEDA, AND T. KIYONO [1980]. "An analysis of Pascal programs in compiler writing," *Software—Practice and Experience* **10**:2, 149-157.

SHUSTEK, L. J. [1978]. "Analysis and performance of computer instruction sets," SLAC Report 205, Stanford Linear Accelerator Center, Stanford University, Stanford, California.

SIPPU, S. [1981]. "Syntax error handling in compilers," Rep. A-1981-1, Dept. of Computer Science, Univ. of Helsinki, Helsinki, Finland.

SIPPU, S. AND E. SOISALON-SOININEN [1983]. "A syntax-error-handling technique and its experimental analysis," *TOPLAS* **5**:4, 656-679.

SOISALON-SOININEN, E. [1980]. "On the space optimizing effect of eliminating single productions from LR parsers," *Acta Informatica* **14**, 157-174.

SOISALON-SOININEN, E. AND E. UKKONEN [1979]. "A method for transforming grammars into LL(k) form," *Acta Informatica* **12**, 339-369.

SPILLMAN, T. C. [1971]. "Exposing side effects in a PL/I optimizing compiler," *Information Processing 71*, North-Holland, Amsterdam, 376-381.

STEARNS, R. E. [1971]. "Deterministic top-down parsing," *Proc. 5th Annual Princeton Conf. on Information Sciences and Systems*, 182-188.

STEEL, T. B., JR. [1961]. "A first version of Uncol," *Western Joint Computer Conference*, 371-378.

STEELE, G. L., JR. [1984]. *Common LISP*, Digital Press, Burlington, Mass.

STOCKHAUSEN, P. F. [1973]. "Adapting optimal code generation for arithmetic expressions to the instruction sets available on present-day computers," *Comm. ACM* **16**:6, 353-354. Errata: **17**:10 (1974) 591.

STONEBRAKER, M., E. WONG, P. KREPS, AND G. HELD [1976]. "The design and implementation of INGRES," *ACM Trans. Database Systems* **1**:3, 189-222.

STRONG, J., J. WEGSTEIN, A. TRITTER, J. OLSZTYN, O. MOCK, AND T. STEEL [1958]. "The problem of programming communication with changing machines: a proposed solution," *Comm. ACM* **1**:8 (August) 12-18. Part 2: **1**:9 (September) 9-15. Report of the Share Ad-Hoc committee on Universal Languages.

STROUSTRUP, B. [1986]. *The C++ Programming Language*, Addison-Wesley, Reading, Mass.

SUZUKI, N. [1981]. "Inferring types in Smalltalk," *Eighth ACM Symposium on Principles of Programming Languages*, 187-199.

SUZUKI, N. AND K. ISHIHATA [1977]. "Implementation of array bound checker," *Fourth ACM Symposium on Principles of Programming Languages*, 132-143.

SZYMANSKI, T. G. [1978]. "Assembling code for machines with span-dependent instructions," *Comm. ACM* **21**:4, 300-308.

TAI, K. C. [1978]. "Syntactic error correction in programming languages," *IEEE Trans. Software Engineering* **SE-4**:5, 414-425.

TANENBAUM, A. S., H. VAN STAVEREN, E. G. KEIZER, AND J. W. STEVENSON [1983]. "A practical tool kit for making portable compilers," *Comm. ACM* **26**:9, 654-660.

TANENBAUM, A. S., H. VAN STAVEREN, AND J. W. STEVENSON [1982]. "Using peephole optimization on intermediate code," *TOPLAS* **4**:1, 21-36.

TANTZEN, R. G. [1963]. "Algorithm 199: Conversions between calendar date and Julian day number," *Comm. ACM* **6**:8, 443.

TARHIO, J. [1982]. "Attribute evaluation during LR parsing," Report A-1982-4, Dept. of Computer Science, University of Helsinki.

TARJAN, R. E. [1974a]. "Finding dominators in directed graphs," *SIAM J. Computing* **3**:1, 62-89.

TARJAN, R. E. [1974b]. "Testing flow graph reducibility," *J. Computer and System Sciences* **9**:3, 355-365.

TARJAN, R. E. [1975]. "Efficiency of a good but not linear set union algorithm," *J. ACM* **22**:2, 215-225.

TARJAN, R. E. [1981]. "A unified approach to path problems," *J. ACM* **28**:3, 577-593. And "Fast algorithms for solving path problems," *J. ACM* **28**:3, 594-614.

TARJAN, R. E. AND A. C. YAO [1979]. "Storing a sparse table," *Comm. ACM* **22**:11, 606-611.

TENNENBAUM, A. M. [1974]. "Type determination in very high level languages," NSO-3, Courant Institute of Math. Sciences, New York Univ.

TENNENT, R. D. [1981]. *Principles of Programming Languages*, Prentice-Hall International, Englewood Cliffs, N. J.

THOMPSON, K. [1968]. "Regular expression search algorithm," *Comm. ACM* **11**:6, 419-422.

TJIANG, S. W. K. [1986]. "Twig language manual," Computing Science Technical Report 120, AT&T Bell Laboratories, Murray Hill, N. J.

TOKUDA, T. [1981]. "Eliminating unit reductions from LR(k) parsers using minimum contexts," *Acta Informatica* **15**, 447-470.

TRICKEY, H. W. [1985]. *Compiling Pascal Programs into Silicon*, Ph.D. Thesis, Stanford Univ.

ULLMAN, J. D. [1973]. "Fast algorithms for the elimination of common subexpressions," *Acta Informatica* **2**, 191-213.

Ullman, J. D., [1982] *Principles of Database ...*, 2nd Ed. Computer Science Press, Rockville, Md.

Ullman, J. D. [1984] *Computational ...*, Vol. II, Computer Science Press, Rockville, Md.

Vyssotsky, V., and P. Wilson [1983] "A graph grammar for advanced graphics analysis," unpublished, AT&T Bell Laboratories, Murray Hill, N.J.

Waxman, J. A. [1971] "A declarative mechanism for regular languages," *Comput. SCI.* 10:3, 363–368.

Weinstein, R. V., and M. L. [illegible] [1974] "Declaring pushing verbs for parsing," *SCIDP* 28:1, 168–172.

Wirth, W. M., [1976a], "Code generation," in Hunter and Tickell [1976], 302–342.

Wirth, W., [1976a] "Optimization," in Hunter and Tickell [1976], 520–602.

Wirth, W. M., and J. R. Clarke [1985] "The use of Lagrange multipliers," *Software — Practice and Experience* 15:3, 227–237.

Wong, W. S. G. [1977] "A Compiler Writer's System with Optimization Capabilities for Complex Data Structure Programs," Ph. D. Thesis, Northwestern Univ., Evanston, Ill.

Woods, D. A. [1977] "The parsing problem for this grammar," *Acta Informatica* 8, 1–23.

Woodward, P. [1974] "The implementation of data types in LIS," *Comput. SCI.* 17:5, 351–364.

Woodward, P. [1975] "Property estimation as well as value propagation," *IEEE Trans. on Software Engineering* 1:3, 290–295.

Williams, M. N. [1983] "Summarizing graphics by regular expressions," *Journal Abstracts of Computer Principles and Programming Languages* 2:3, 203–216.

Williams, M. N., and P. K. Zadrozny [1985], "Constant propagation with conditional branches," *Journal Abstracts of Computer Principles and Programming Languages*, 201–209.

Williams, J. F. [1981] "Notes on Algol 68," *Newsletter* 1981, pp. 120–127.

Wood, A. J. [1980] "Three-pass column data flow analysis to the presence of pointers, procedure variables, and label variables," *Second Annual ACM Symposium on Principles of Programming Languages*, 53–64.

Woodward, W. W. [1976] *Logical Systems*, *Studies in Logic and the Foundations of Mathematics* [E. J. Thesis, Yale University, New Haven, Connecticut.

Wright, J. W. J. [illegible] [1974] "Ambiguities and recursion in Pascal," *Software — Practice and Experience* 7:6, 665–676.

Wallis, S., R. L. [1981] *Dynamic Programming Languages*, Academic Press, New York.

White, D. [1981] "PL/W — a program writing language for the 8086 microprocessor," *SCIDP* 15:1, 37–54.

Wirth, N. [1971] "The design of a Pascal compiler," *Software — Practice and Experience* 1:4, 309–333.

Wirth, N. [1981] "Pascal-S: A subset and its implementation," in Barron [1981], pp. 199–259.

Wirth, N., and H. Weber [1966] "Euler: a generalization of Algol and its formal definition: Part I," *Comm. ACM* 9:1, 13–23.

Wirth, D. [1968] "The theory of well-behaved languages," *Computer J.* 12:4, 349–356.

Wolf, W., A. R. K. Johnson, L. B. Wilson, K. N. O. Herbst, and U. M. Gehring [1975] *The Design of an Optimizing Compiler*, American Elsevier, New York.

Yamashita, M. [1985] Private communication.

Yu, Sook D. M. [1967] "Recursive data processing in the C programming language," in Lunar, J. C. (ed.), *Computing Surveys* 10:2, 189–215.

Zilles, K. N., and W. G. Bush [1972] "Optimization of structured programs," *Software — Practice and Experience* 4:1, 51–57.

ÍNDICE ALFABÉTICO

A

[illegible]

[illegible]

[illegible]

B

[illegible]

C

Cálculo de Custo Unitário de Produção, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186
Calendário, 163
Caminhão, 201, 202, 203
Capital de Giro, 89, 90, 91
Capital Social, 45, 46, 47
Carga Horária, 112, 113, 114
Carteira de Trabalho, 67, 68, 69
Centro de Custo, 134, 135, 136
Certificado de Conclusão, 198
Cheque, 54, 55, 56
Cidade Universitária, 210, 211, 212
Cliente, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50
Comissão de Formatura, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259,

Cooper, [illegible] 205, 208–209, 214–217
  division of 137–142, 143
  evidence of Late Neolithic origin
  sites of 137–138
Copper-arsenic alloys 25, 28, 45
Copper-tin alloys 260–261, 301
Copper mines, Italian 215, 220, 227
  abandoned 228
Corbels 43, 57, 60–61, 132
  function of 62
  development of 52
  decoration of 57, 129
  symbolism of 61
Corinth, Gulf of 1, 3, 5, 7
Cornwall 251
Corporate identity
  and burial customs 201–202
  of Beaker groups 183–184
  and [illegible] 302
  and pottery 180–181
  and ritual sites 202
Crete 178
Crop-marks 132
  cursus 132
  Grimes Graves 133
  sites revealed by [illegible] 135, 138, 139, 140
  and [illegible] 143
  Stonehenge 111, 113, 115, 118
  and [illegible] 143
  and Silbury 148–149
Cruciform type, 275, 278, 283, 286, 301
Cyclades 18
  Cemetery at 111
  pottery 17, 19, 20, 21, 22, 108
  pottery 111
  [illegible] 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75,

Kosmose, A. C., 118
Kawamura, S. R., 31
Kirshner, K., 143
Kurtz, C. H. S., 48
Kail, [illegible]
Kirchner, B. H., 119
K[illegible], H., 253
Kossal, [illegible]

L

Lace, 251, 252, 263, 266, 269, 288
 [illegible] 263
 [illegible]
 [illegible]
 [illegible] 268
 [illegible] 269
[illegible] 310
[illegible] 253, 254, 261
[illegible] 28
Lachlan, W. R., 18
Lewis, D. A., 251
Lemaire, R. W., 190
Lamb, P. T., 191
Langmuir, H., 71
Lassigue, T., 253
Lattice, [illegible] 26
Lemaire, C., 118
Legrand, H. L., 165
Lemon, R. R., 119
Lemaire, T., 115
Leck, M. E., 50, 80
Lesson, R. W., 221, 253
Levy, [illegible], 253
Levy, [illegible] 190
Lewis, [illegible]
Lewis, P. M., 15, 137
Lexie, 46, 55, 58, 62, 70, 101
Levison, 25, 27, 35, 98, 98
Lurie, 230
[illegible] 15
[illegible] 11, 48, 52, 75, 76, 90
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
LISCHST, 140
Lint, 123
Lisson, 12, 46, 98, 303, 317
Lisa
 [illegible] 52
 [illegible] 186, 187, 190
Lisse, [illegible]
[illegible], 5, 90, 94, 100
Lerner, 8, 116
Lux, [illegible]
Lewis, [illegible] 11, 318, 322
Lucas, P., 52
Lund, [illegible]
Lunnel, H., 253

M

MacLean, M. D., 318
MacQueen, J. B., 161, 168
Mann, R., 25, 190
Mason, C. M., 148
[illegible] 18, 25
[illegible] 191
[illegible]
Mapa de memória, 192
[illegible]
[illegible]
[illegible]
 [illegible] 28, 31, 280, 287
Martin, T., 146, 188
Markman, J., 318
Martello, S., 169
Marr, 265, 266, 271, 323
Mattes, [illegible]
Maxwell, W. L., 18
Mayer, D. H., 148
McArthur, H., 221, 318
McCarthy, J., 37, 98, 315
McLennan, R. M., 118
McCracken, S. J., 145
McGilliam, W. S., [illegible]
McLane, M., 21
McKeown, W. M., 37, 118, 100, 253
McLellan, H. R., 253
McNaughton, R., [illegible]
Mealey, C. W., 137, 253, 303, 312, 318, 322
Mecheri, T., 158
Memória, 130
Mensagem de [illegible] 86, 93, 94, 110, 111
MIL-TX, 118
Miller, [illegible]
Mayer, R., 65
Miller, R., [illegible]
Milner, R., 132, 160
Minimização de estados, 64, 66
Minsky, T., 251
Mitchell, R. C., 251
Mitchell, [illegible]
ML, 136, 139, 165
Modelo de [illegible] 71, 71, 96, 110
Modelo de [illegible]
Modula, 2[illegible], 323
Módulo [illegible] 11
[illegible] 301
Modelo de [illegible] 9
Montanari, U., 169
Moore, E. F., 23, 318
Moran, [illegible] 51
Morel, [illegible]
Morris, D., 25
Morris, J. S., 75, 110
Morris, R., 190
Morse, N. P., 118
Moses, J., 190
Muller, D. E., 119
Movimento [illegible] 256, 258, 288, 303, 314, 325
Muchnick, S. S., 168, 318
MUX, 140
Muller, M. E., 118

N

Nakata, [illegible]
Naur, P., 37, [illegible], 168, 169
NEATS, [illegible]
[illegible]
Newey, M. C., 22
Nievergelt, J., 253
Nó
 [illegible] 248
 [illegible] 165, 166
 [illegible] 288
 [illegible] 288
Nome, 187
 [illegible] 118, 139
 global, 288
 [illegible]
 [illegible] 177
 [illegible] 169, 177, 188, 189
Nori, K. V., 221
Número de [illegible] 103
Número de [illegible] 125, [illegible]
Não terminal, 13, 75, 90
Número [illegible] 135, 145

O

O'Donnell, M. J., 255
Objeto, 167, 169
Ogden, W. F., 116
Operador
 [illegible] 258
 [illegible] 272, 297, 318
 [illegible] 287
 [illegible] 92
Ordem
 [illegible] 123, 128, 135, 148
 [illegible]
Ordenação topológica, 127, 288, 303, 304
[illegible]
[illegible] 170, 172
 [illegible] 264, 290
[illegible]
[illegible]
 [illegible] 258, 264, 268, 272, 280, 288, 294, 303, 311
[illegible] 124, 308
[illegible] 250, [illegible]
 [illegible]
 [illegible] 250
 [illegible] 250, 253, 258

P

P-code, 121, 128
Pager, D., 119
Pai, A. B., 158
Paige, R., 315
Pair, C., 146
Palavra, 12
 [illegible] 25, 70, 88
 reservada, 25, 90
Palm, R. C., 115
Panini, 52
[illegible] 221, 248
[illegible]
 [illegible]
 [illegible] 176, 180
 [illegible] 67
Parênteses, 45, 47, 95
Park, J. C. H., 119
Partição, 64
 de [illegible] 81
Pascal, 25, 39, 43, 44, 71, 140, 150, 156, 157, 171, 180, 182, 186, 204, 208
Passagem, [illegible]
 de [illegible]
Paterson, M. S., 160
PDP, [illegible] 322
Pennello, T., [illegible]
[illegible] 288
[illegible] 68
[illegible]
[illegible]
Persch, G., 116
Peterson, W. W., 191, 318
Peyolle, [illegible], 318
Pig, 150
Pike, R., 73, 190, 319, 328
Pilha, 57, 65, 85, 93, 118, 125, 127, [illegible], 208, 209, 213
 [illegible] 171
[illegible] 161
Pixley, W., 20
PL, [illegible] 273
PL/I, [illegible] 73, [illegible] 188, 211, 220, 221, 318
Plankalkül, 158
[illegible] 165
[illegible] 67, 82
Pollack, B. W., 31
Pollock, [illegible] 318
[illegible]
[illegible] 268
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
Powell, M. L., [illegible] 325
Pozefsky, D., 116
Pratt, [illegible] 190
Pratt, V. R., 70, 118
[illegible] 268
[illegible] 28
[illegible] 15, 47, 91, 101, 104, [illegible]
 [illegible]
 [illegible] 118
 de operadores, 11
 [illegible]
 [illegible]
[illegible] 280
Prefixo, 12
 [illegible] 88, 95, 96, 100
PRIMEIRO, 91, 87, 84, 89
[illegible]
 [illegible] 178, 182, 205, 206
[illegible]
 [illegible] 78, 134, 145
 [illegible] 78, 84, 146
 [illegible]
 [illegible]
Produto
 [illegible] 138
 [illegible] 11
Profundidade

[illegible]

Q

[illegible]

R

[illegible]

S

[illegible]

T

[illegible]

U

[illegible]

V

[illegible]

[illegible]

W

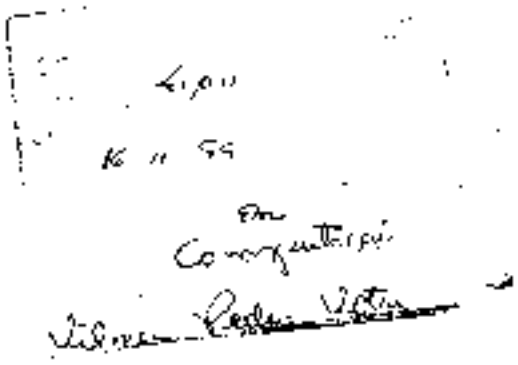
[illegible]

Y

[illegible]

Z

[illegible]



Serviços de impressão e acabamento
para obras e peças de [illegible]
[illegible]
Tel.: (21) 98[illegible]-[illegible] | [illegible]
[illegible]